

Success in Acquisition: Using Archetypes to Beat the Odds

William E. Novak
Linda Levine

September 2010

TECHNICAL REPORT
CMU/SEI-2010-TR-016
ESC-TR-2010-016

Acquisition Support Program
Unlimited distribution subject to the copyright.

<http://www.sei.cmu.edu>



This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2010 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. This document may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about SEI publications, please visit the library on the SEI website (www.sei.cmu.edu/library).

Table of Contents

Acknowledgments	v
Abstract	vii
1 Introduction	1
1.1 The Problem	1
1.2 Background and Rationale	2
2 Systems Thinking	5
2.1 Feedback Loops and Open and Closed Systems	5
2.2 Causal Loop Diagrams	6
3 The Systems Archetypes	9
3.1 Fixes That Fail	10
3.2 Shifting the Burden	11
3.3 Accidental Adversaries	13
3.4 Escalation	15
3.5 Drifting Goals	16
3.6 Growth and Underinvestment	18
3.7 Success to the Successful	20
3.8 Limits to Growth	22
3.9 Tragedy of the Commons	23
3.10 Balancing Loop with Delay	25
3.11 Some Observations on Systems Thinking and the Systems Archetypes	26
4 Applying the Systems Archetypes to Software Acquisition	29
5 The Acquisition Archetypes	31
5.1 The Bow Wave Effect	31
5.2 Firefighting	35
5.3 Everything for Everybody	38
5.4 Feeding the Sacred Cow	41
5.5 PMO Versus Contractor Hostility	45
5.6 Staff Burnout and Turnover	49
5.7 Underbidding the Contract	52
5.8 Longer Begets Bigger	55
5.9 Robbing Peter to Pay Paul	58
5.10 “Happy Path” Testing	61
5.11 Brooks’ Law	64
5.12 Shooting the Messenger	68
6 Challenges, Implications, and Future Directions	71
6.1 Short-Term Thinking	72
6.2 Misaligned Goals	74
6.3 Future Directions	75
References	79

List of Figures

Figure 1:	Causal Loop Diagrams of Reinforcing and Balancing Loops	7
Figure 2:	Causal Loop Diagram of “Fixes That Fail”	10
Figure 3:	Causal Loop Diagram of “Shifting the Burden”	11
Figure 4:	Causal Loop Diagram of “Accidental Adversaries”	13
Figure 5:	Causal Loop Diagram of “Escalation”	15
Figure 6:	Causal Loop Diagram of “Drifting Goals”	16
Figure 7:	Causal Loop Diagram of “Growth and Underinvestment”	18
Figure 8:	Causal Loop Diagram of “Success to the Successful”	20
Figure 9:	Causal Loop Diagram of “Limits to Growth”	22
Figure 10:	Causal Loop Diagram of “Tragedy of the Commons”	24
Figure 11:	Causal Loop Diagram of “Balancing Loop with Delay”	25
Figure 12:	Causal Loop Diagram of “The Bow Wave Effect”	33
Figure 13:	Causal Loop Diagram of “Firefighting”	37
Figure 14:	Causal Loop Diagram of “Everything for Everybody”	39
Figure 15:	Causal Loop Diagram of “Feeding the Sacred Cow”	43
Figure 16:	Causal Loop Diagram of “PMO vs. Contractor Hostility”	47
Figure 17:	Causal Loop Diagram of “Staff Burnout and Turnover”	50
Figure 18:	Causal Loop Diagram of “Underbidding the Contract”	53
Figure 19:	Causal Loop Diagram of “Longer Begets Bigger”	56
Figure 20:	Causal Loop Diagram of “Robbing Peter to Pay Paul”	59
Figure 21:	Causal Loop Diagram of “‘Happy Path’ Testing”	62
Figure 22:	Causal Loop Diagram of “Brooks’ Law”	65
Figure 23:	Causal Loop Diagram of “Shooting the Messenger”	69

Acknowledgments

Many people have contributed to creating this report, both directly and indirectly. It would not have been possible to discuss such a wide range of software acquisition topics without the insights, expertise, and prior work of others. We would also like to thank a key sponsor, the Department of Veterans Affairs, for the original opportunity to perform this work, and our other customers and sponsors who enabled us to work on technical engagements that stimulated and contributed to the ideas expressed here.

We extend our thanks to people within the Carnegie Mellon[®] Software Engineering Institute (SEI) and beyond for their comments and reviews of early versions and for their support of this work: Joe Elm, John Foreman, Brian Gallagher, Michael Goodman, Patricia Oberndorf, Robert Rosenstein, and Ray Williams.

Finally, the authors are most grateful for the essential help we have received from our editors, Gerald Miller and Barbara White, and our graphic designer, Melissa Neely. Their efforts have resulted in a better document.

[®] Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Abstract

This project on patterns of failure is based on experiences with actual programs and employs concepts from *systems thinking* to analyze dynamics that have been observed in software development and acquisition practice. The software acquisition and development archetypes, based in part on the general systems archetypes, have been created as part of an ongoing effort to characterize and help manage patterns of counterproductive behavior in software development and acquisition.

This report introduces key concepts in systems thinking and the general systems archetypes, and then applies these concepts to the software-reliant acquisition domain. Twelve selected software acquisition and development archetypes are each described and illustrated by a real-life scenario, and guidance is provided on both recovering from and preventing these dynamics. Finally, the authors consider implications of the work and future directions for research.

1 Introduction

This project on patterns of failure is based on experiences with actual programs and employs concepts from *systems thinking* to analyze dynamics that have been observed in software development and acquisition practice. The software acquisition and development archetypes have been created as part of an ongoing effort to characterize and help manage patterns of counterproductive behavior in software development and acquisition. These archetypes use the ideas of systems thinking to describe common patterns of failure so that they can be anticipated and prevented.

In this report we begin by describing key elements in systems thinking. We proceed with an introduction to the general systems archetypes, and then apply these concepts to the software acquisition domain. Twelve selected software acquisition and development archetypes are described. Each is illustrated in an actual scenario, and guidance is provided on both breaking and preventing these dynamics. Finally, we consider implications and directions for the future—for research and for the use of acquisition archetypes in the field.

1.1 The Problem

Perhaps the most puzzling question in the software and systems acquisition world is this:

Why do problems persist in software development and systems acquisition, despite the fact that solutions to many of these problems exist and have been known for decades?

For 50 years, federally funded research and development centers, think tanks, and advisory bodies—such as the MITRE Corporation, the Aerospace Corporation, the RAND Corporation, and the Defense Science Board—have analyzed barriers and enablers to the acquisition and development of software-intensive systems. For example, the following challenges for policy and topics for research were set down almost 30 years ago [Dews 1979]:

- Improve the acquisition information data base.
- Reduce the instability in program funding and scheduling.
- Strengthen guidance on hardware competition in development.
- Emphasize production quantity as an element in the requirements process.
- Continue offering incentives to make program management an attractive service career.
- Examine the timing of program manager (PM) assignments.

This list of topics remains pertinent, although an updated version would highlight additional challenges, including software development and deployment, security, and system-of-systems interoperability. It is disturbing, however, that many long-standing problems associated with the development and acquisition of software-intensive systems remain unresolved—and are growing in magnitude—while proposed solutions remain either untried or have not been sustained. In this report, we consider why this is so and ask: What alternative perspectives from systems thinking might break this logjam?

1.2 Background and Rationale

Countless panels have studied the acquisition process in the past 20 years. Speaking in 2005 about the Pentagon's panel review of acquisition, Chairman Ronald T. Kadish (a retired Air Force three-star general who previously headed the Missile Defense Agency) said: "The perception is that no reforms have addressed systemic weaknesses in structure, process, and governance of acquisitions" [Merle 2005].

In concurrence, Danielle Brian, executive director of the Project on Government Oversight, a watchdog group, remarked: "I think there are many large buildings filled with reports from commissions about reforming the Pentagon's acquisition system. It's time for people to have to make the hard decisions rather than putting them off by just studying the issue to death" [Merle 2005].

Certainly, no single factor can explain the pervasive and persistent problems in systems acquisition or the inclination to study matters rather than to take action. Multiple factors and dimensions contribute to the current state of acquisition practice and its challenges: political, technological, legal, economic, and cultural. Nonetheless, it is troubling that problems persist where solutions are known. Similarly, there is little consolation in excusing a situation by observing that the problems are "systemic" in nature and therefore too hard to remedy.

Compounding the persistence of decades-old problems is the increasing complexity we see in all aspects of systems and organizations. The solutions that we demand are complex, requiring extraordinary levels of coordination, cooperation, and collaboration. This can only be accomplished by reaching beyond the usual silos and stovepipes to which we are accustomed and attending to joint responsibilities, interdependencies, and interoperability.

In 1991, in *The Fifth Discipline*, Peter Senge wrote:

Perhaps for the first time in history, humankind has the capacity to create far more information than anyone can absorb, to foster far greater interdependency than anyone can imagine, and to accelerate change far faster than anyone's ability to keep pace. Certainly the scale of complexity is without precedent [Senge 1991].

Now, nearly 20 years later, Senge's observation reads like an understatement. There is widespread agreement on increasing complexity and its acceleration in technical and social systems, reinforcing the need for new models for organizational structures and capabilities. Nowhere is this better expressed than in the vision of net-centricity. The net-centric vision requires increased richness and reach simultaneously and promises to connect our store of advanced combat and intelligence platforms and provide them with timely, accurate data [Evans 2000]. Margaret Myers, Deputy CIO for the Department of Defense (DoD), explains: "Net-centricity allows users to augment data available from their own systems and capabilities with data from other locations and other sources well removed from the normal sensor range of the platform itself. In the net-centric environment, global information and local data sources will be fused to provide what we call 'power to the edge'" [Myers 2002].

Even more modest examples represent a challenge, as linear behaviors become *nonlinear* and seemingly unpredictable when combined. For example, the interactions between a program management organization (PMO), contractor, subcontractors, sponsors, and users in an acquisition organization are complex and nonlinear—producing behavior that appears unpredictable and unmanageable.

An additional factor exacerbates the challenge of managing complexity. To date, we lack effective problem-solving methods that address dynamic complexity and serve a *whole systems* view [Senge 1991]. Conventional analytical methods that “break things apart” via decomposition and division of labor fail to attend to the interdependencies, relationships, and interfaces—in other words, to the interstices in technical and social systems.

“Divide and conquer” approaches represent old forms that are top-down, partitioned, and fraught with assumptions of independence. These tools and methods are appropriate for handling *detailed complexity* where there are many variables. However, Senge elaborated on a second type of complexity: *dynamic complexity*. This is the type of complexity that we are concerned with here. Dynamic complexity refers to “situations where cause and effect are subtle, and where the effects over time of interventions are not obvious.” When the same action has dramatically different effects in the short run and the long run, there is dynamic complexity. When an action has one set of consequences locally and a very different set of consequences in another part of the system, there is dynamic complexity. When obvious interventions produce non-obvious consequences, there is dynamic complexity [Senge 1991].

Nonetheless, traditional problem-solving strategies that serve detailed complexity are all too often applied in attempts to solve problems of dynamic complexity. We know, for example, that assumptions of independence of subcomponents are rarely, if ever, true—given our understanding of coupling—and yet we often conveniently ignore the effects of coupling and integration. We proceed as if things can be broken apart and then just added back together again without introducing unintended consequences or emergent behaviors—as if the “whole is not greater than the sum of the parts.” Aggregation, interactions, feedback, and interdependencies are largely unaccounted for or dismissed.

The following scenario illustrates this plight. We are demanding more and more of our systems. We are relying on problem-solving models and old methods of decomposition (for detail) because they’ve always served us well in the past—and we assume and hope that they will continue to do so. We have no better alternatives readily available, even if we suspect that our current approaches have weaknesses and limitations. In fact, “simulations with thousands of variables and complex arrays of detail can actually distract us from seeing patterns and major interrelationships” [Senge 1991].

Our organizations reflect the systems that we develop and the problem-solving strategies that we use to analyze and develop those systems—and in turn, the systems that we develop influence our organizational constructs. In both technical and organizational systems we have rich legacies that are now proving to be inadequate for current challenges. We believe that these conditions of increasing complexity, virtuality, and interoperability are ripe for new problem-solving models,

for new approaches to dynamic complexity, for abstracting, handling, and transforming information.

Systems thinking approaches, including systems archetypes, are holistic in nature. Every influence is both cause and effect. Rather than reinforcing linear thinking, a “parts” mentality and analytical models of decomposition and detail, true systems thinking places emphasis on feedback, influence, and interdependencies. In this regard, it holds promise for understanding and succeeding in the realm of dynamic complex systems.

2 Systems Thinking

When we try to pick up anything by itself we find it is attached to everything in the universe.

—John Muir

Why is systems thinking important? Because in a world where we are increasingly recognizing that many of our most serious challenges are stemming from our inability to manage complex, nonlinear, dynamic systems, we can take more efficient and effective steps to address our problems by attempting to understand both the unintended and the longer term consequences of our decisions. One manifestation of the need for systems thinking is that people are inclined—and sometimes encouraged by those with political or economic interests—to believe in *causal* relationships based on only *correlational* information. Where the root cause of a complicated problem is not fully understood, but nonetheless must be addressed, solutions based on loose, correlational data may be advocated in the hope that they will produce results that affect the root cause. Some examples of this pattern include eating oat bran to reduce the chance of heart disease, putting more police officers on the street as the most effective way to reduce crime, putting babies to sleep on their backs to avoid Sudden Infant Death Syndrome (SIDS), and so on. People *want* to believe that there are straightforward solutions to complex and difficult problems, and in the absence of a proven remedy they are likely to adopt this simplified way of thinking. The use of systems thinking helps to avoid this trap, by looking for the unintended consequences of proposed solutions in order to find if they are in actuality “quick fixes” that will *not* resolve the problem satisfactorily, and may in fact spawn *new* problems that are worse than the original.

Another indication of the need to change our thinking lies in the distinction between *first-* and *second-order change*. First-order change calls for doing more of the same to address an issue—increasing the application of the current remedy. Second-order change calls for making a change to the *structure* of the system that is creating the issue. Second-order change requires acknowledging that the current remedy is no longer working and stepping back from the situation to reassess options; this often results in trying a different approach altogether. Systems thinking techniques support this change in perspective and provide both insight and guidance when applied to our most complex issues. As we shall see in this report, attempts at first-order change that are conducted within the existing structure often exacerbate rather than resolve the issue they were intended to address.

2.1 Feedback Loops and Open and Closed Systems

Systems thinking is a method for analyzing complex systems. It has its roots in system dynamics work pioneered by Jay W. Forrester at MIT in the 1960s [Forrester 1971]. Forrester recognized that the behaviors of the electrical feedback loops he was studying—loops that would either amplify or diminish a signal or regulate and balance it—were in many ways similar to patterns of behavior he had observed within organizations and even globally. Systems thinking views systems as sets of components that have complex interrelations occurring between them, many of

which take the form of feedback loops in closed systems. These systems occur commonly in economics, the environment, business, politics, and organizations of all kinds.

The world is filled with open and closed systems. A heater with an “On-Off” switch is an *open* system. It continues to heat the room regardless of how hot it becomes. A heater with a *thermostat* is a *closed* system, because it has a feedback loop that senses the room temperature and shuts off the heater if it exceeds a preset value. However, in practice, even a simple heater is part of a closed system, because a *person* acts as the sensor and then operates the switch, creating a feedback loop.

Feedback loops are of two types: *reinforcing* (i.e., positive feedback) loops and *balancing* (i.e., negative feedback) loops. Reinforcing loops tend to continually increase or continually decrease. Balancing loops ultimately converge on, or oscillate around, a stable equilibrium at some value. These will be discussed in more detail in the following section.

2.2 Causal Loop Diagrams

There are many different analytical tools used in the application of systems thinking [Kim 1998]. They include behavior over time (BOT) diagrams, causal loop diagrams (CLDs), and “stock and flow” diagrams. Each of these techniques has a different focus and purpose. BOT diagrams describe the behavior of specific system variables over time, to help the analyst interpret the significance of the behavior from a systems thinking perspective. Causal loop diagrams illustrate the values and feedback loops present in a system and how they interact with one another. Stock and flow diagrams provide more formalism, which is necessary to describe behaviors so that they can be simulated by a computer. A “stock” is the term for an entity that increases or decreases over time. A “flow” refers to the rate of change in a stock. In this report, due to their concise, qualitative nature, we focus on the use of causal loop diagrams as the “tool of choice” for communicating the high-level structure of closed systems in software development and acquisition.

Causal loop diagrams depict the dynamic causes and effects in a situation by showing how variables (represented by nodes) relate to and influence one another (represented by arrows). The effect of the arrows is labeled “S” for “Same” (i.e., when one variable increases, the next variable increases as well—or when one variable decreases, so does the other) or “O” for “Opposite” (i.e., when one variable goes up, the next one declines, or vice versa). The arrows can come together to form loops (see Figure 1). These loops are labeled either “B” for “Balancing,” describing loops that converge toward a stable equilibrium (where x increases, y decreases), or “R” for “Reinforcing,” describing loops that continually or even exponentially increase or decrease (where x increases, y increases). The term “Delay” in the diagram indicates actual time delays that occur.¹ Such time delays are very significant to a human operator attempting to control a system, as they can obstruct the ability to clearly see the connections in cause-and-effect relationships.

¹ Note that the presence of a time delay is not inherent in either a reinforcing or balancing loop, but rather can be used as needed.

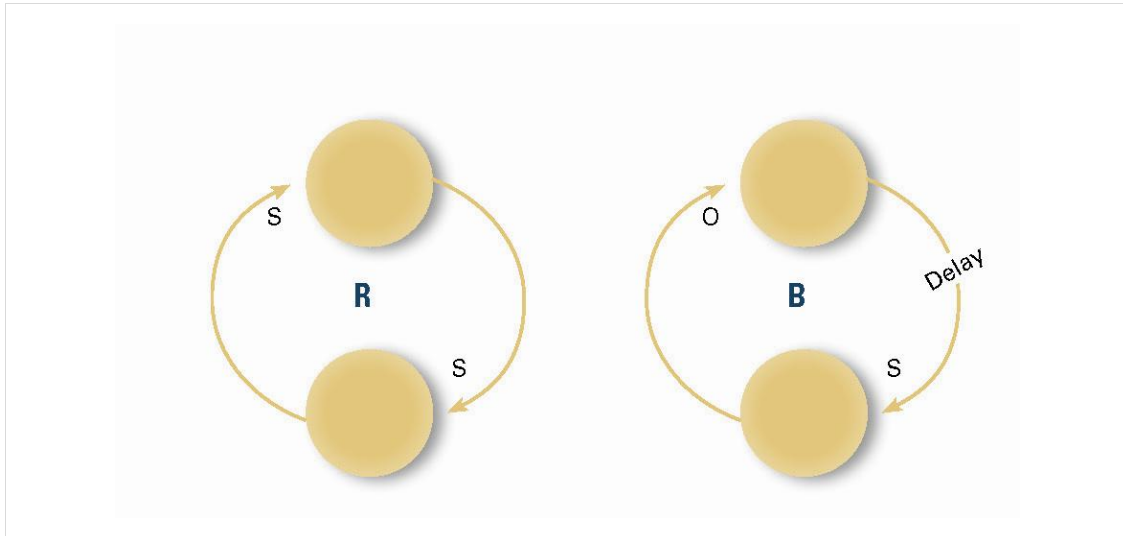


Figure 1: Causal Loop Diagrams of Reinforcing and Balancing Loops

A loop can be identified as being either reinforcing or balancing based on the number of “Same” and “Opposite” arrows within the loop. Loops with an even number of “Opposite” relationships are reinforcing, whereas those with an odd number are balancing.

We can think of reinforcing loops as “engines” that drive a system, causing either growth or decline. Balancing loops are those that provide stability or equilibrium—they are the “controls” that regulate a system. Both kinds of loops are needed. There are many examples of both reinforcing and balancing feedback loops in everyday life. A familiar reinforcing loop is illustrated by compound interest, which continues to reinforce itself by increasing in value based on the amount by which it has already increased, creating exponential growth. Other examples of reinforcing loops include population growth and the spread of an epidemic. An example of a balancing loop would be real estate (and other economic) cycles that oscillate back and forth, seeking a comparatively steady state.

3 The Systems Archetypes

Stories or accounts are most commonly used to communicate patterns and lessons, and give them authenticity. However, the use of stories can have an unfortunate side effect, should the pattern become tied to or lost in the details of the specific story, making it difficult for people to generalize the key message and apply it in other fundamentally identical situations. The systems archetypes address this problem by describing the basic form of the pattern in terms of a causal loop diagram, using evocative and descriptive names to communicate the essence of the dynamic.

The systems archetypes each describe a generic story, a scenario that plays out in many different situations and environments, but always follows the same underlying pattern. Despite the prevalence of these storylines, there is still some surprise on the part of those who are swept up in the dynamics of each of the systems archetypes—a feeling of, “There I was, just doing my job like I always have, when out of the blue, through no fault of my own, I got sideswiped by *this*—and now I don’t know how to get out.” Usually, this is the result of the “side-effect” or the “unintended consequence” of the archetype. Resolving these patterns, once they’re set in motion, can’t be accomplished by doing more of the same thing that has been done before. Just as “doing what you always do” can set an archetype in motion, it often requires doing something counter-intuitive or unexpected to break the pattern—because the archetypes do not resolve themselves.

Ten “classic” systems archetypes have been identified although work continues both to add to this initial set, and also to further condense it to a smaller set that displays only the most significant structural differences [Rahn 2005, Haraldsson 2005, Wolstenholme 2003]. In this section we present brief overviews of the original nine systems archetypes described by Senge in *The Fifth Discipline*, and a tenth called “Accidental Adversaries” that was introduced later in *The Fifth Discipline Fieldbook* [Senge 1991, 1994]:

- Fixes That Fail
- Shifting the Burden (a.k.a. “Addiction”)
- Accidental Adversaries
- Escalation
- Drifting Goals
- Growth and Underinvestment
- Success to the Successful
- Limits to Growth
- Tragedy of the Commons
- Balancing Loop with Delay

For each archetype we include an opening statement of the intent of the systems archetype, a detailed description of the flow of the dynamic, its causal loop diagram, and one or two examples of the pattern in real-world situations. We conclude this section with observations on systems thinking and the systems archetypes.

3.1 Fixes That Fail

A quick fix for a problem has immediate positive results, but its unforeseen long-term consequences worsen the problem.

Description

“Fixes That Fail” (sometimes called “Fixes That Backfire”) begins with the *Problem Symptom*. As the *Problem Symptom* increases, a *Fix* must be applied to address it. The *Fix* in turn alleviates the *Problem Symptom*, creating a balancing loop. However, the *Fix* also has *Unintended Consequences* that exacerbate the problem. As the *Fix* is applied more frequently, more *Unintended Consequences* occur—and the increasing *Unintended Consequences* that emerge after some time delay result in the return of the original *Problem Symptom* or its worsening.

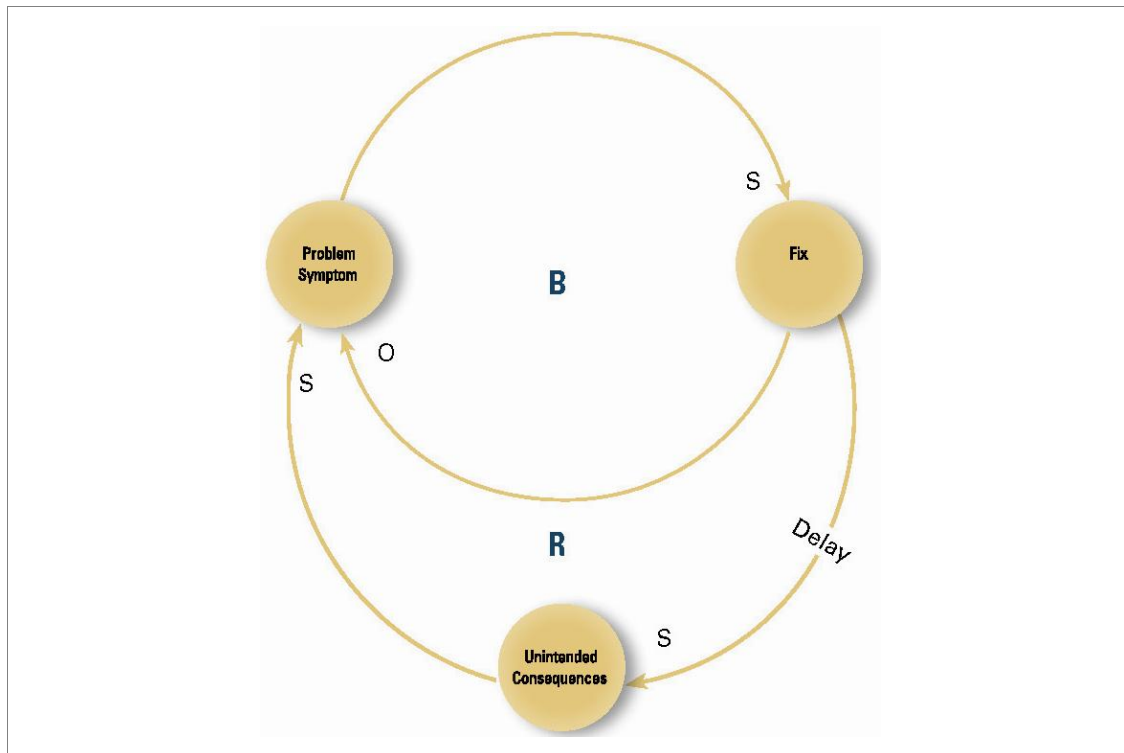


Figure 2: Causal Loop Diagram of “Fixes That Fail”

Examples

- Using a credit card to pay off debt, which temporarily alleviates the problem, but then worsens the total debt through additional interest from finance charges
- Increasing hiring to augment existing experienced staff, but then finding that the experienced staff’s time is largely consumed by bringing the new hires up to speed, resulting in a sharp loss in productivity

3.2 Shifting the Burden

An expedient solution temporarily solves a problem, but its repeated use makes it more difficult to use the fundamental solution. “Shifting the Burden” is also known as “Addiction,” because use of a symptomatic solution can become addictive, even though the side-effects are ultimately damaging.

Description

“Shifting the Burden” also starts with a *Problem Symptom*. A choice exists between applying the *Symptomatic Solution* or the *Fundamental Solution* to address the *Problem Symptom*. The *Fundamental Solution* has a significant time delay before it has an effect on the original *Problem Symptom*, which leads to a preference for using the more immediate *Symptomatic Solution*. The problem is supposedly solved by using the *Symptomatic Solution* (B1) and attention is diverted away from a more *Fundamental Solution*. Greater application of the *Symptomatic Solution* causes a decrease in the original *Problem Symptom*, keeping it in balance. However, increasing use of the *Symptomatic Solution* causes more of the unintended *Side Effect* to be produced, which in turn decreases the ability to use the *Fundamental Solution*—making the organization more dependent on the *Symptomatic Solution*, and ultimately trapping it into using only the *Symptomatic Solution*.

If the *Fundamental Solution* had been chosen, as the *Problem Symptom* increased, the use of the *Fundamental Solution* would have also increased, and after a time delay there would be a decrease in the original *Problem Symptom*, again keeping them in balance.

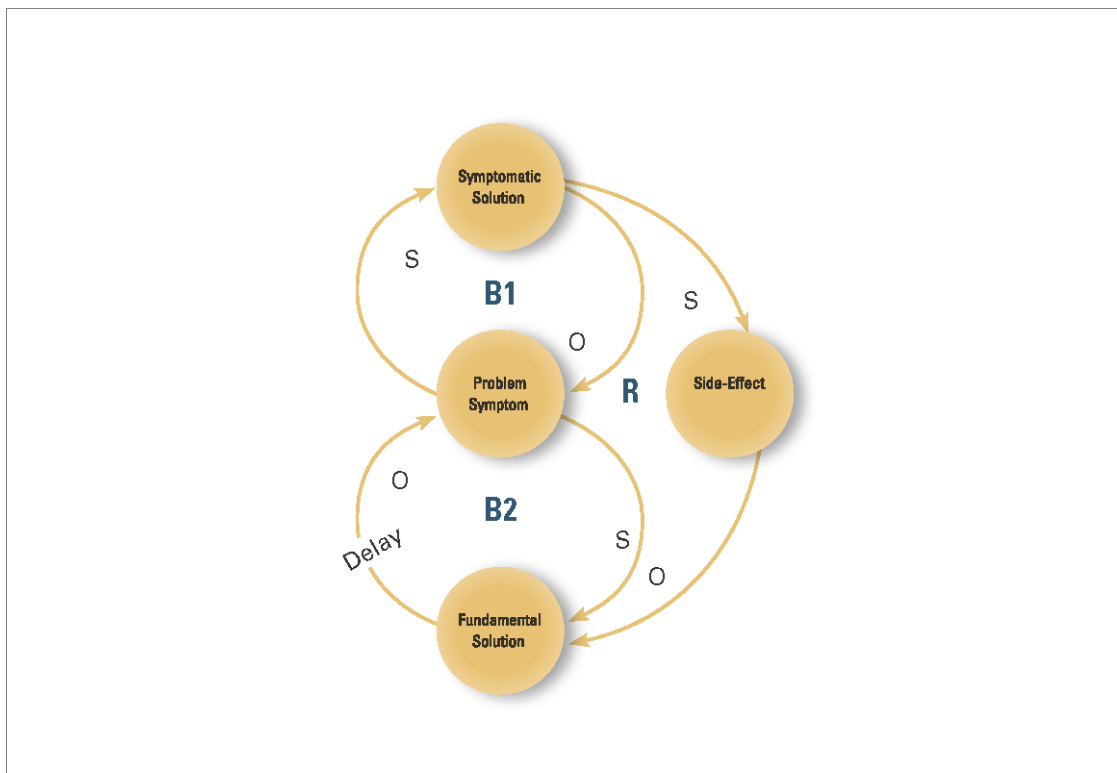


Figure 3: Causal Loop Diagram of “Shifting the Burden”

Examples

- An addiction to caffeine, which temporarily alleviates sleepiness, but has an unintended side effect, over time undermines the ability to employ the fundamental solution—getting more sleep
- Increasing the short-term profitability of a factory by cutting back on maintenance, counting on being promoted due to high near-term profits, before production begins to fall off

3.3 Accidental Adversaries

Two parties destroy their relationship through escalating retaliations for perceived injuries.

Description

The “Accidental Adversaries” systems archetype describes a pattern in which the outermost reinforcing loop of *A’s Activity Toward B* → *B’s Success* → *B’s Activity Toward A* → *A’s Success* is undermined by actions that A and B take individually to help themselves. At the same time A is taking action to improve its own position independently of B, creating a reinforcing loop in which *A’s Activity Toward A* increases *A’s Success*, which in turn leads to more of *A’s Activity Toward A* to further increase *A’s Success*. However, *A’s Activity Toward A* turns out also to negatively impact *B’s Success*, and B in turn implements *B’s Activity Toward B*, which then turns out to adversely impact *A’s Success*. While the original intent is simply to take actions to improve your own position, the result is that they (at first unexpectedly) adversely impact your partner’s position—and thus are perceived as being deliberate and malicious, and the partner responds (or retaliates) in kind.

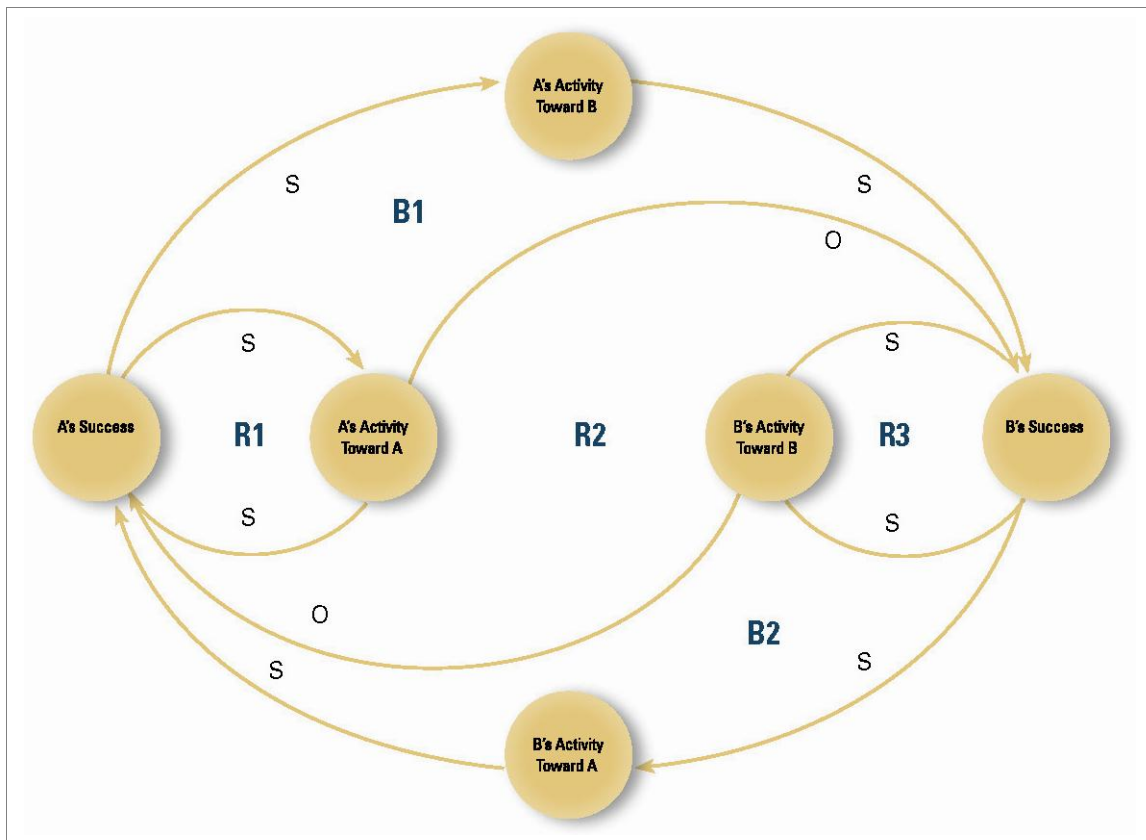


Figure 4: Causal Loop Diagram of “Accidental Adversaries”

Examples

- A failed marriage, in which unintentional actions such as sloppiness can be perceived as being deliberate and offensive by the partner and can then escalate in the form of retaliations until the relationship ends in divorce
- A fast-food company expands outlets through the use of franchisees that have to maintain standards set by the parent company, but the parent company also has its own outlets. As the company expands its own outlets to improve profitability, it moves into markets perceived by franchisees as belonging to them, resulting in lawsuits and a loss in popularity of the line of fast food.

3.4 Escalation

Two parties compete for superiority, with each escalating its actions to get ahead.

Description

“Escalation” describes the situation in which two competitors take increasingly extreme actions to achieve superiority. At the core of escalation are two actors (perhaps more) who feel a sense of threat by the actions of the other. Each actor endeavors to keep things under control by managing its own balancing process. More *Activity by A* improves *A’s Result* and improves the *Quality of A’s Position Relative to B’s*. This then decreases the *Threat to A* that is posed by B and decreases further *Activity by A*. By itself this is a balancing loop, but when coupled with an identical balancing loop on B’s side, it creates a large reinforcing loop. Imagine “untwisting” the two loops (by duplicating the center node), and what emerges is a single large reinforcing loop causing A and B to continually ratchet up, or *escalate*, their activities toward one another.

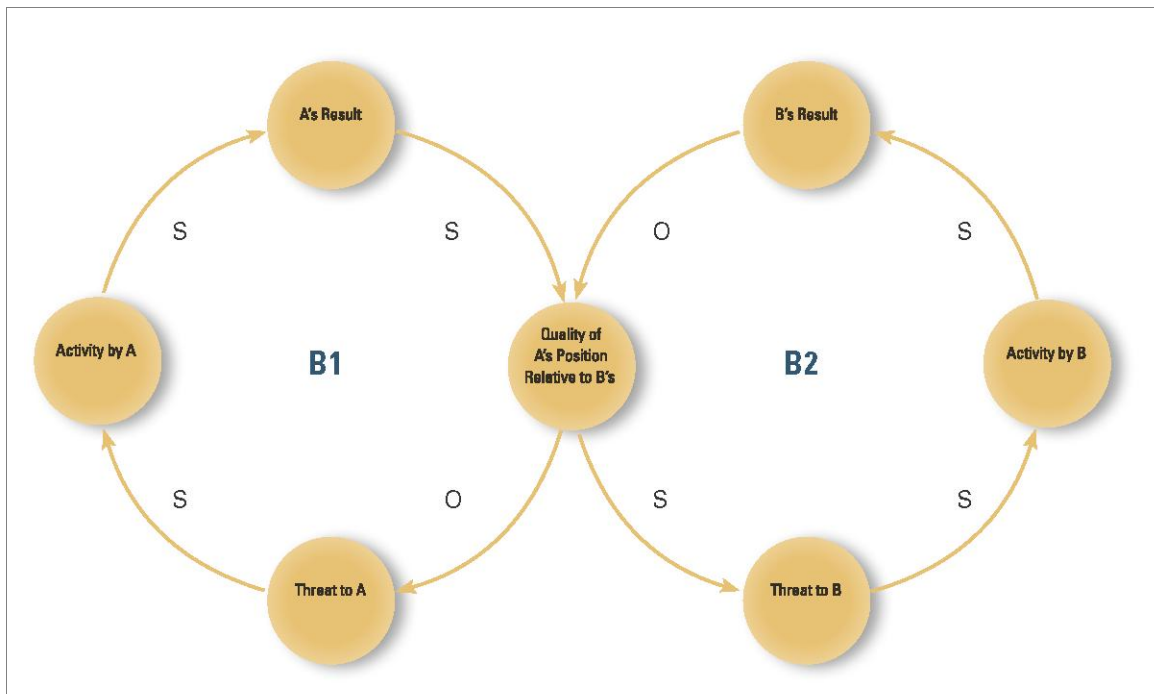


Figure 5: Causal Loop Diagram of “Escalation”

Examples

- The nuclear arms race, in which one country’s efforts to surpass another’s nuclear arsenal, simply spurs the other on to greater efforts to increase its *own* stockpile
- A price war between two similar businesses, where the efforts of one business to undercut the prices of the other and gain market share lead the other business to respond in kind

3.5 Drifting Goals

A gradual decline in performance or quality goals goes unnoticed, threatening the long-term future of the system.

Description

“Drifting Goals” (sometimes called “Eroding Goals”) describes the tension between two balancing loops that represent two competing pressures. A gap between the goal and the current state can be addressed by taking corrective action (B1) or by lowering the goal (B2). In a business setting, this may be illustrated through slipped deadlines so that a once inconceivable delay of four weeks becomes acceptable and routine. Likewise, quality reviews and standards previously held in high regard can be lowered in order to address either a backlog or schedule delay. The upper loop shows the ongoing pressure to lower the organization’s own goals (or high standards), while the lower loop shows the result of making improvements in the product or service offered by the organization in order to achieve its goal. In the upper loop, a high *Goal* widens the *Gap* between *Actual* performance/quality and the *Goal*, which increases *Pressure to Lower Goal*, which eventually lowers the *Goal*. In the lower loop, as the *Actual* performance/quality of the organization declines, the *Gap* is increased, spurring more *Corrective Action*, which, after a delay, improves the *Actual* performance/quality of the organization. It is the time delay of the *Corrective Action* that makes it attractive to resort to the more expedient action of lowering the goal instead.

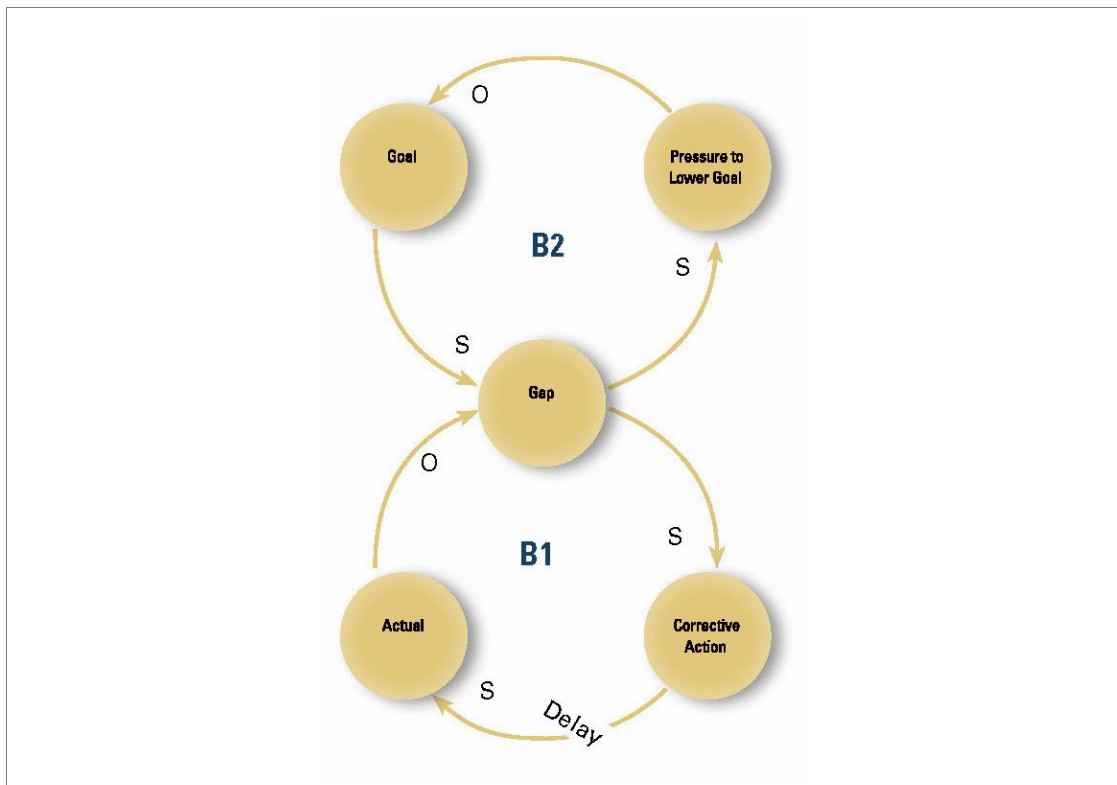


Figure 6: Causal Loop Diagram of “Drifting Goals”

Examples

- Gradually replacing high-quality ingredients with lower quality (and lower cost) substitutes—corn syrup for sugar, shortening for butter, artificial flavorings instead of real ingredients—has expedient ways of reducing cost, and thus reducing the gap between actual profits and desired profits, instead of (a) finding more cost-effective ways of obtaining or producing those ingredients or (b) investing in more sophisticated marketing of the product so that the product can justify a higher price to cover the increased costs
- Repeatedly “rebaselining” a program’s cost and schedule to be more expensive and longer because the initial estimates (on which the government approved the investment in the program in the first place) are seen to be unachievable as the program progresses

3.6 Growth and Underinvestment

Investments in a growing area aren't made, so growth stalls, which then becomes the rationale for further underinvestment.

Description

The “Growth and Underinvestment” systems archetype consists of three loops. In the first loop (upper left), as the organization’s *Growth Effort* increases, *Demand* also increases in a reinforcing loop. However, in the second loop (middle) as *Demand* increases, the organization’s *Performance* must also increase to keep pace with the *Demand* (a classic “supply and demand” relationship)—and by satisfying that *Demand*, it declines in a balancing loop. The third loop (lower right) describes the organization’s *Capacity*, and shows a balancing loop in which current *Performance* is compared to an existing *Performance Standard*. As *Performance* declines, the *Perceived Need to Invest* increases, so *Investment in Capacity* is increased, and after a delay, *Capacity* increases. Increased *Capacity*, in turn, then increases *Performance*. Again, it is the time delays that ultimately make it easier to reduce the *Growth Effort* than to make the required *Investment in Capacity*.

We will see that “Growth and Underinvestment” has at its heart a “Limits to Growth” (or “Limits to Success”) archetype. The extra loop (B2) illustrates how failing performance can be used to justify underinvesting in the very capacity that is necessary to avert the limit to growth.

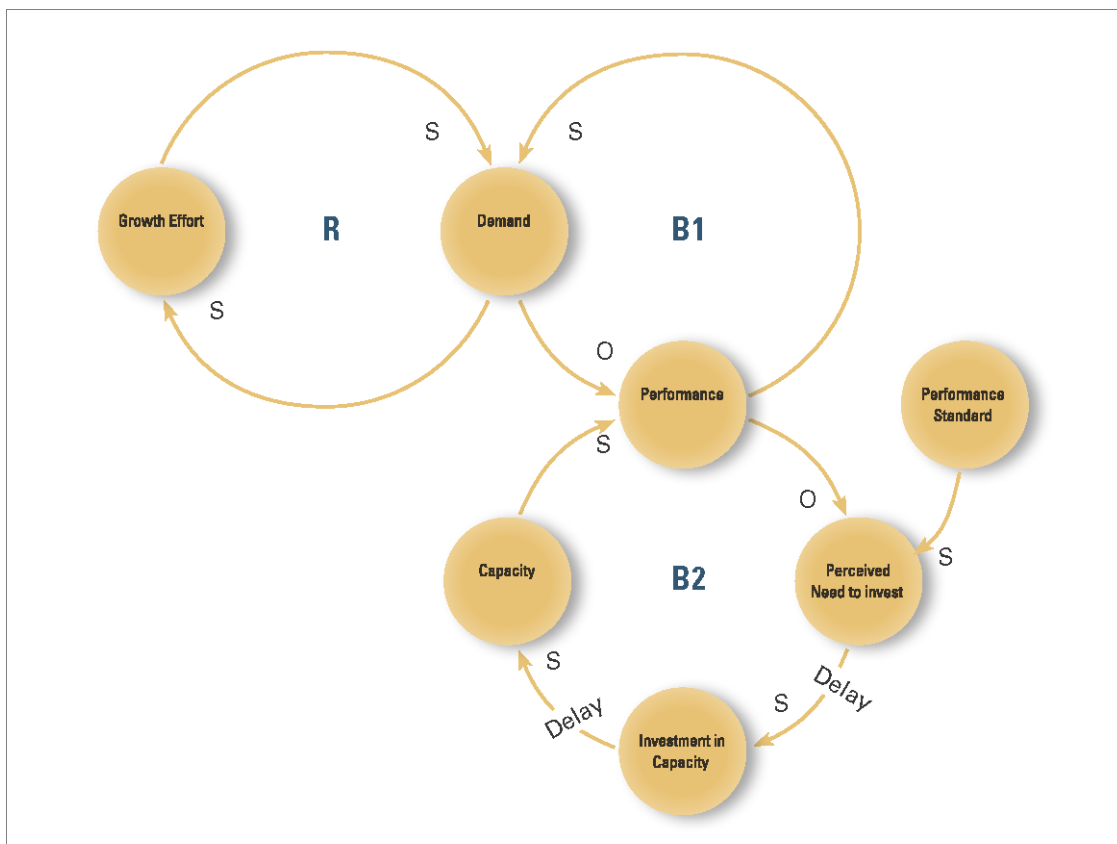


Figure 7: Causal Loop Diagram of “Growth and Underinvestment”

Examples

- The demise of People's Express airline is widely believed to be due to a failure to grow the customer service function so that it would be able to keep pace with the growth of the rest of the airline.
- Trying to learn to play the piano without a teacher saves money in the short run by underinvestment, but the desired proficiency is never achieved, leading to unfulfilled expectations, disillusionment; interest in practicing gradually fades.

3.7 Success to the Successful

When two parties compete for a limited resource, the initially more successful party receives more of the resource, increasing its success at the expense of the other, thus making it more likely to *continue* to receive more of the resource.

Description

Success does not always come from talent; in fact, just as often success may be a consequence of structure. The “Success to the Successful” dynamic consists of two almost identical and joined reinforcing loops representing two actors or organizations. In each loop the center node (perhaps an overarching manager) divides the resource between the two competing organizations. For example, if organization A receives the majority of the allocation, that increases the *Resources to A*, thus increasing the *Success of A*, and ultimately continuing to increase the *Allocation to A Instead of B*, thus *further* increasing A’s allocation of the shared resource—and the cycle continues. For the party receiving the *minority* of the allocation, the loop moves in the opposite direction: *Resources to B* decline, which slows down the *Success of B* and *increases* the *Allocation to A Instead of B*. A then continues to receive more of the resource and experiences more success, while B continues to receive fewer resources and experiences less success. Once the balance tips in favor of one or the other, since the equilibrium is unstable, it continues to tip farther and farther in that same direction.

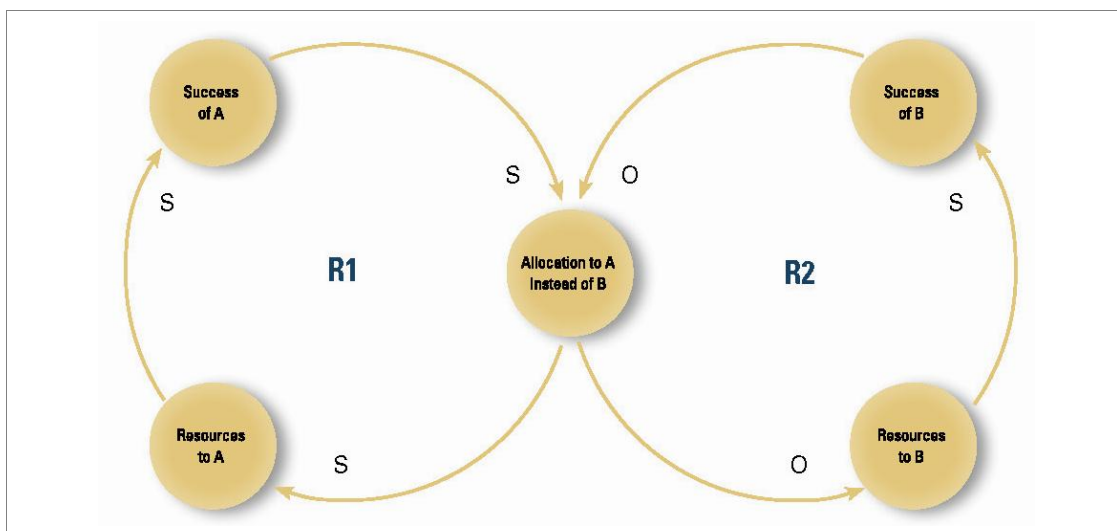


Figure 8: Causal Loop Diagram of “Success to the Successful”

The “Success to the Successful” archetype is highly sensitive to initial conditions—in other words, like pushing a snowball down a hill, with the two interacting reinforcing loops, it only takes a very small push to produce large differences in outcomes later on. The “Butterfly Effect” captures this concept of the sensitivity of a system to initial conditions, a key notion from chaos theory. The idea, first noted by Edward Lorenz while studying weather systems, is that small differences in the initial conditions (i.e., the flapping of a butterfly’s wings) of a nonlinear dynamic system (i.e., the weather system) can produce large changes in the long-term behavior of the system (i.e., global weather patterns) [Lorenz 1963].

Examples

- The ascendancy of the VHS video format over Betamax, in which a format with greater capacity per tape ultimately triumphed over a technically superior format because of the initially small advantage of the VHS standard. By quickly surpassing each increase in Betamax storage capacity, VHS ultimately captured the entire market.
- Incoming students who have high standardized test scores (e.g., intelligence) may get more attention from instructors, providing these students with greater incentives to work hard and excel in subsequent standardized tests.

3.8 Limits to Growth

An initially rapid growth slows because of an inherent capacity limit in the system that worsens with growth.

Description

“Limits to Growth” (sometimes called “Limits to Success”) consists of a reinforcing loop linked to a balancing loop. The reinforcing loop describes a classic growth engine—more *Efforts* produce better *Performance*, and in turn better *Performance* spurs on even greater *Efforts*. However, this growth engine is linked to a balancing loop that limits it—as *Performance* increases, some *Limiting* (i.e., slowing) *Action* occurs, based on a *Constraint* that exists in the organization or the environment, that reduces *Performance*. Thus, despite an organization’s increasing efforts, it is unable to drive its *Performance* past a certain point that is imposed by the *Constraint*. The real breakthrough in “Limits to Growth” lies in identifying and removing the *Constraint* that is limiting growth while there are still sufficient resources to do so.

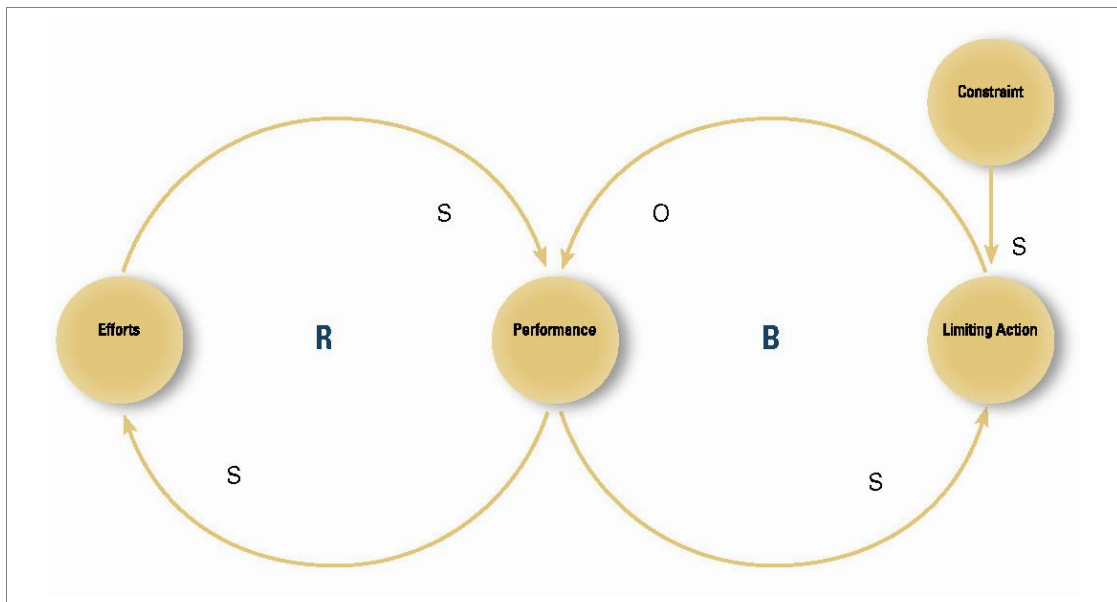


Figure 9: Causal Loop Diagram of “Limits to Growth”

Examples

- A suburban town becomes a popular place to live because of its bucolic surroundings, peaceful atmosphere, and close-knit populace. As more and more people move there, new housing developments replace the quiet countryside, traffic congestion fills the streets, and the rapidly increasing population introduces large numbers of strangers—and people gradually stop moving there.
- A successful software contractor wins a large government contract and increases software programming staff to execute it—only to find that they have already hired all the good programmers in the immediate area, and that the local universities do not produce qualified graduates at a high enough rate to satisfy their needs.

3.9 Tragedy of the Commons

A shared resource is depleted as each party abuses it for individual gain, degrading or destroying the resource, ultimately hurting all who share it.

Description

In this archetype each actor pursues actions that are individually beneficial, but that eventually culminate in a situation that is worse for all involved. When the individual gains and the activities of all become too large for the system to support, the commons becomes overloaded and everyone experiences diminishing benefit. “Tragedy of the Commons” contains numerous loops and so appears to be especially complex—but the underlying concepts and processes are not difficult to understand.² As we saw in “Accidental Adversaries,” the top and bottom reinforcing loops are analogous growth engines, showing how *A’s Activity* produces more *Net Gains for A*, which in turn leads to more *A’s Activity* to further increase *Net Gains for A*. However, because the “commons” is a shared resource, the activities of A and B are not independent, but are instead tightly linked, as shown in the center reinforcing loops. Here we see that as *A’s Activity* and *B’s Activity* increase, the *Total Activity* increases—but due to the *Resource Limit* inherent in the shared resource this means that, after a delay, the net *Gain Per Individual Activity* decreases. When this occurs, A and B are both forced to increase their levels of activity, presumably to make up for the shortfall in expected gain. At the same time the decline in *Gain Per Individual Activity* also decreases the *Net Gains* for each party. In short, too much activity by the parties involved (who make use of the shared resource) eventually starts to deplete the resource, spurring the parties on to even greater activity to make up the shortfall, which only depletes the resource more, further reducing their *Net Gains*.

² The term “commons” originally referred to areas of shared grassland where peasants were allowed to graze their animals.

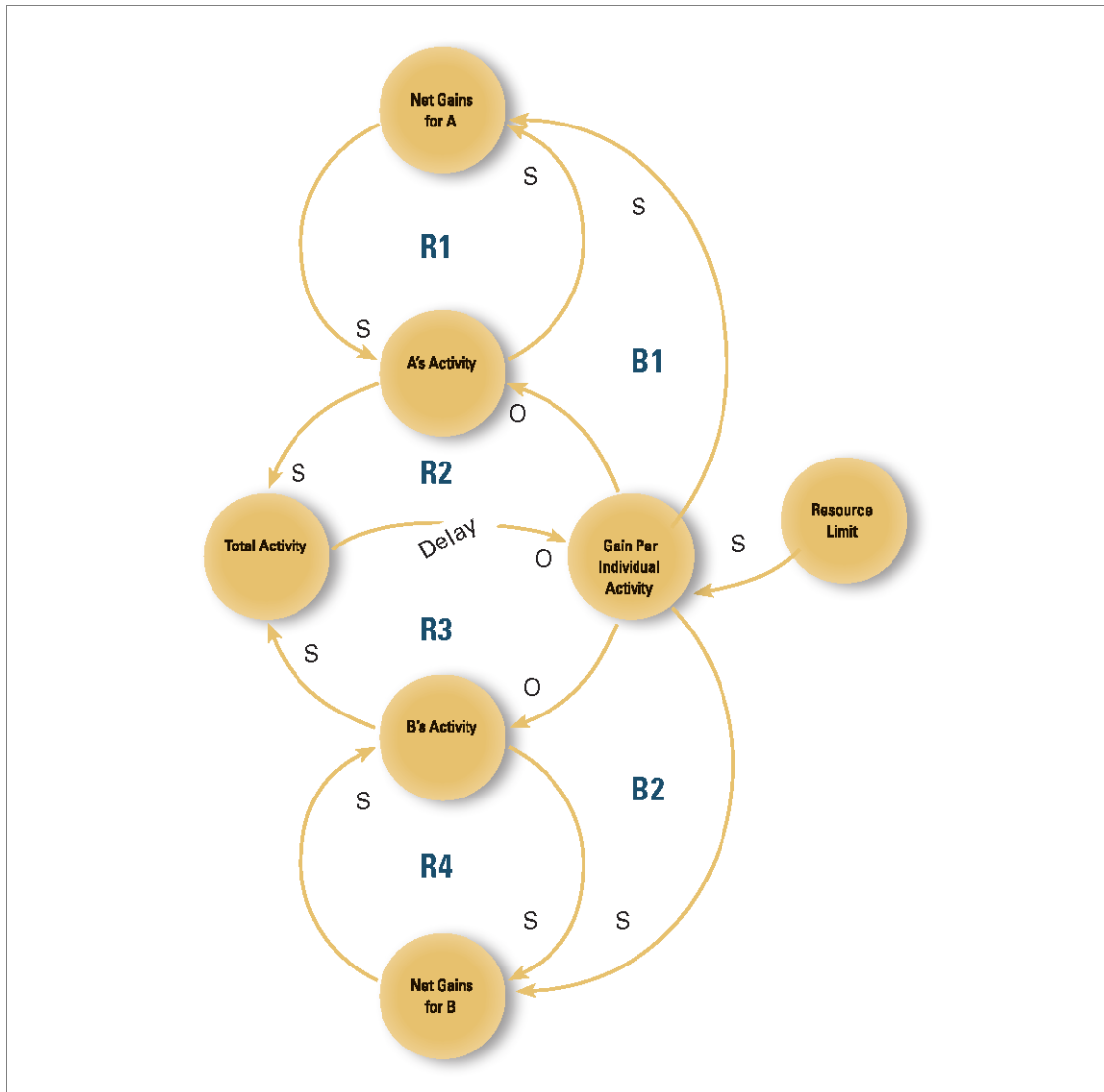


Figure 10: Causal Loop Diagram of "Tragedy of the Commons"

Examples

- Over-fishing of popular fishing grounds, in which the fish population becomes substantially depleted due to the steadily increasing efforts of commercial fishing boats to capture more of the declining catch
- Air pollution, where producing more airborne pollutants allows greater production and revenue for an individual company, but at the expense of the air quality for all
- Reserving conference rooms in an organization where they are in chronically short supply can lead to groups starting to over-schedule the rooms, exploiting this shared resource by reserving more rooms farther and farther in advance without having a specific need for them, in order to avoid the inconvenience of not having one on the day it may truly be needed

3.10 Balancing Loop with Delay

The current state of a system is moved toward the desired state through repeated action, but the delay raises doubts about its effectiveness.

Description

“Balancing Loop with Delay” is one of the simplest of the structures, but it is also one of the most important because of its wide application. In this archetype there is a *Gap* between the *Current State* and the *Desired State*. As the *Current State* declines, the *Gap* between it and the *Desired State* increases. As the *Gap* increases, *Action* must be taken to attempt to close the *Gap*. As more *Action* is taken, after a delay, the *Current State* improves, narrowing the *Gap* and bringing the system back toward a balanced steady state (no *Gap* between the *Current State* and the *Desired State*). However, the time delay makes it difficult to see if the *Action* has produced sufficient results, causing additional (and unnecessary) application of the *Action*—and if the *Action* taken is too extreme, the *Gap* “flips” in the opposite direction after the time delay due to the overcorrection.

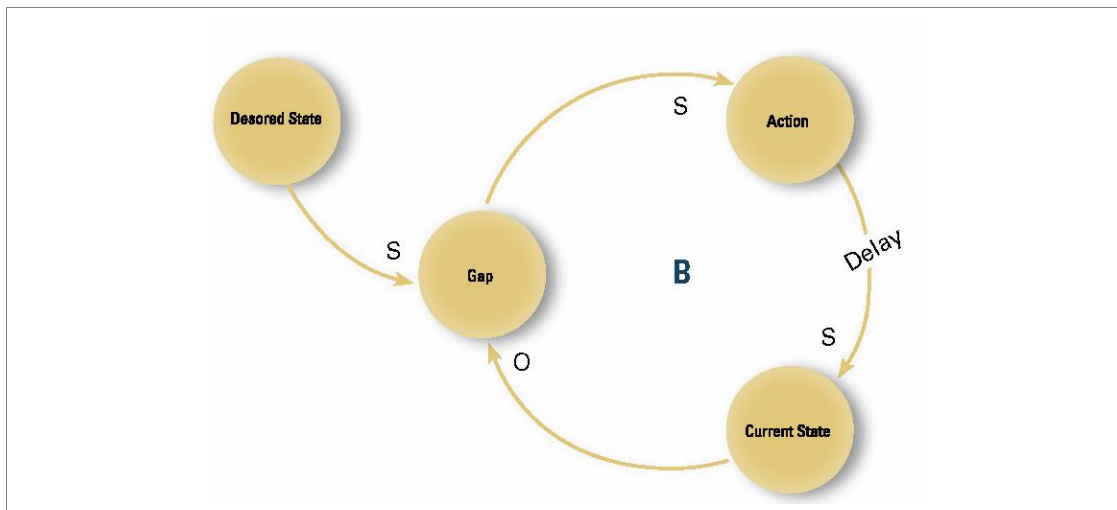


Figure 11: Causal Loop Diagram of “Balancing Loop with Delay”

Examples

- Over-steering a large vessel that is slow to respond (resulting in weaving back and forth), a common failure of novices that is only overcome with training and experience
- A thermostat and furnace/air conditioner that may require a substantial amount of time to change the temperature of a house, especially if the requested temperature change is large—causing oscillations between the house being too hot and too cold as the homeowner impatiently overcorrects the thermostat setting

3.11 Some Observations on Systems Thinking and the Systems Archetypes

The sections below offer observations on the use of systems thinking, the systems archetypes, and causal loop diagrams. These are presented here to provide some insights that may be useful in subsequent discussions involving the application of these concepts to software acquisition and development.

Shifting Loop Dominance, or Finding the “Tipping Point”

A key concern for those who closely examine a systems archetype such as “Fixes That Fail” is determining when (or whether) it will be the case that the quantitative impact of the *unintended* (negative) consequence will be enough to overwhelm the *intended* (positive) effect. This is sometimes referred to as “shifting loop dominance” because the loop that is dominating the dynamic’s behavior will change over time. For example, a stable balancing loop may, over time, be overwhelmed by a reinforcing loop that continues to grow—or the growth of a reinforcing loop may eventually be constrained by the limit imposed through a balancing loop. The point at which a shift in loop dominance occurs is often referred to as a “tipping point.” Unfortunately, qualitative systems thinking approaches cannot answer the question of *when* the tipping point is reached, although the application of quantitative system dynamics can be used to provide this [Repenning 2001, Ford 2005]. However, system dynamics still requires that many assumptions be made regarding the precise nature of the quantitative relationships between the values involved. Even if it proves impossible to accurately characterize these relationships, a system dynamics simulation can provide useful qualitative conclusions about the general behavior of the causal loop structure.

“Rate-to-Level” Flows

There are some well-known inherent formal weaknesses in using causal loop diagrams. So notes George Richardson in discussing a causal loop diagram showing the effect of migration on a local population by linking *migration* to move in the same direction as *population*:

Consider the link from migration to population. The definition claims that a change in migration will produce a change in population in the same direction, yet a decrease in migration will not produce a decrease in population unless migration becomes negative, drawing people out of the city. As long as migration is positive, it will always increase the population of the community, whether migration itself is increasing or decreasing. Furthermore, it is not even always true that an increase in migration produces an increase in population... [Richardson 1986].

The issue here is that migration is a *rate*, and not a *quantity* (i.e., a “level”) like population. Simply because the *rate* of change in a variable starts to decline does not necessitate that the absolute quantity it influences *also* starts to decline. In a more general sense, this means that even if the semantic logic seems to work in a causal loop diagram, the dynamic pattern may not. Richardson refers to these problematic links as “rate-to-level” links (i.e., those that attach a *rate* to a *level*) or “conserved flows.” The issue is that rate-to-level links can cause the standard characterizations of positive (reinforcing) and negative (balancing) loops to be false. This puts an

additional burden on the designer of causal loop diagrams. While one way to address this is to use “stock and flow” diagrams instead, this affects the ability of many people to read and understand the dynamics being illustrated—thus largely defeating the illustrative purpose behind causal loop diagrams.

Reversibility

As the diagram is traced out from node to node, it is implied that time is passing—and when a “delay” is added, it becomes explicit. Having said that, the systems archetypes all have “reversibility” as one of their properties—that is, they make logical sense when traversed in either direction (increasing over time, or decreasing over time). This is *not* a required attribute of causal loop diagrams, although it can be useful. It is possible to create causal loop diagrams that are not reversible. For example, if values such as “Remaining Schedule,” “Work Completed,” “Sunk Cost,” or “Knowledge of the System” are used, the diagram cannot be reversible because each of these values can move only in one direction as time passes. They all represent values that either accumulate or drain, similar to a “stock” in a system dynamics “stock-flow” diagram. This means that these kinds of values *cannot* be used in a balancing loop, because in that structure they would be forced to alternately both increase *and* decrease as iterations are made through the loop with the passage of time.

Loop Topology

All causal loop diagrams, by their nature, consist of a combination of linked balancing and reinforcing loops showing the relationships among a set of values. Two causal loop diagrams may have the identical structure of nodes, loops, and arrows (i.e., the same number of reinforcing and balancing loops connected in the same way) and yet, by changing only the variable names, can describe *different* dynamics. Similarly, it is possible to describe essentially the *same* dynamic using *different* loop structures. This is true because causal loop diagrams do not possess formal rigor, and for this reason they are primarily used as a tool for high-level analysis and communication of dynamic structures.

Motivation

The systems archetypes have little or no explicit connection to the motivations of the actors involved in the scenarios they present. We do not know, and the archetype does not state, if an action is *accidental*, *unintentional*, or *misguided*, as opposed to being *deliberate* or *malicious*. The archetype is focused on the *behavior* rather than on the motivation *behind* the behavior, and the cascade of effects from that behavior is the same regardless. The archetypes are thus appropriate for evaluating actions and behaviors in a neutral fashion, and identifying corrective and preventative actions that could be taken. This can make recovering from a recognized dynamic in an organization less confrontational since assigning blame is not helpful in analyzing the situation and is in fact often counterproductive.

4 Applying the Systems Archetypes to Software Acquisition

When a project begins, no one *intends* to deliver it late, or to overrun their budget, or to give users an unreliable system. It just seems to happen all too often, and despite the best of intentions. However, the problems that plague so many software acquisition efforts are predictable—which means that they are also avoidable, and often correctable.

Acquisition organizations are dynamic systems, where the interactions between the PMO, the contractor, subcontractors, sponsors, and users exhibit feedback and thus are complex and non-linear—producing behavior that appears to be unpredictable and unmanageable. Beneath this unpredictability, however, are common structures that drive these behaviors, and these common structures can be understood and managed. For example, when difficulties occur in an organization such as a software acquisition program, the instinctive reaction is to address the *effects* that are causing the immediate pain to the program. This amounts to dealing with the symptom, rather than the underlying problem—and once the immediate pain is relieved, interest in dealing with the root cause is often forgotten. Left unchecked, this natural and intuitive response is not only ineffective in the long term, but can erode the program’s ability to solve the fundamental issue causing the problems.

A systems thinking approach offers the chance to identify such dysfunctional behaviors, gain insights into the root causes of problems, and design interventions that can be used to manage, stop, and prevent such behaviors. Thus, the intent of creating software acquisition and development archetypes is to

- **Identify common dysfunctional behaviors and their causes.**

A substantial part of any improvement or problem-solving effort can be identifying that there is a problem in the first place—and if there is, what that problem is, and what might be causing it. A set of common archetypes provides a useful starting point for making that determination, and provides insight into the most likely causes for those problems.

- **Promote shared understanding of problems.**

By using an explicit representation of the behavior, there can now be a common understanding of the dynamic across the organization. Once there is a shared understanding, or “mental model” of the dynamic, it is possible to address potential solutions without the concern that people may be trying to solve different problems without being aware of it.

- **Engage in “big picture” thinking.**

Thinking about a problem in its larger context may be key to resolving it successfully because the “side-effects” and “unintended consequences” of potential solutions may only be visible in that context. Systems thinking thus helps to improve decision making by avoiding oversimplification. This kind of thinking is essential as software acquisition programs increasingly tackle such challenges as systems-of-systems (SoS), interoperability, and emergent behavior.

- **Diagnose failure patterns as they develop.**

A model of the typically hidden interactions between the many components of a complex system can be used as an “early warning system” to help managers and practitioners identify the leading indicators of acquisition and software development failure patterns.

- **Identify interventions to break out of ongoing dynamics.**

A model of the dynamic can be used to define high-leverage interventions to help break out of classic acquisition failure patterns by leveraging the underlying structure. Knowing which parts of the structure are contributing to the growth and/or decline of the issue at hand helps to make clear where leverage can best be applied to slow the dynamic or even change its direction entirely. Some examples of these kinds of structural interventions are:

- Change a negative dynamic into a positive one by running the archetype backwards.
- Stop feeding an unwanted reinforcing loop by acting to minimize its acceleration, or actively slowing its growth (or decline)—in other words, “When you’re in a hole, stop digging.”
- Change the limiting value around which a balancing loop is oscillating, or which it is approaching, to something more acceptable.

- **Prevent future counterproductive behaviors or dynamics from developing.**

Being familiar with a set of counterproductive behaviors that are common to software acquisition and development provides an essential first step toward preventing them. For each dynamic there are specific actions that can be taken to prevent or reduce the likelihood of its occurrence.

The following sections describe the results of this application of systems thinking to software acquisition and development, and the specific archetypal behaviors that analysis has identified.

5 The Acquisition Archetypes

This section consists of 12 Acquisition Archetypes published by the SEI between 2007 and 2010. All of the Archetypes can be found on the SEI's website.³

5.1 The Bow Wave Effect

A Never-Ending Project

This is a true story—and one you've probably heard before. That's the point. It's about a pattern of failure, an archetype.

A government program needed to replace an aging COBOL mainframe financial system—one so old that the costs of maintaining its obsolete hardware multiplied each year. The only people who could maintain it were now retiring, taking their knowledge with them. Yet the replacement project was stuck in low gear: time dragged on, the focus of the program shifted, deadlines were missed. The sponsors became increasingly anxious. It had, as one team member said, "... drifted, moved, and waddled, and done everything but die."

Finally, with the CIO under increasing political pressure to show IT results, the absolute, final deadline was set—just 18 months away.

How Bow Waves Begin

Could the development team get it done? Yes—but only if they kept to schedule and stuck to the project plan. And that's not what happened. Instead, the *bow wave* pattern of failure stalked the project almost from the start. Fueled by the accumulated effects of an educated guess (SWAG) estimation process, the project picked up baggage, rather than momentum.

"[Requirements] were prioritized, and they got SWAGs, and they drew a line based on available resources," lamented a team member. "They approved [requirements] before they were costed. Some things moved from release to release if they fell below the priority line."

*It was a 3-year program
in its 13th year.*

This practice of deferral sent ripples through the project. It wasn't done maliciously or even consciously by the project teams. Quite the opposite. The effect was the end result of accumulated decisions that seemed right and expedient at the time.

The project managers didn't recognize the problem, or understand that the bow wave is, unfortunately, a common pattern in software development programs. Deferred or dropped

³ The 12 Acquisition Archetypes are located in the Acquisition Support section of the SEI's website (www.sei.cmu.edu/acquisition/research/archetypes.cfm).

functionality and system requirements accumulate, piling up in front of the project in a wave that washes up over schedules and budgets, endangering delivery and project success.

Often, as in this project, the bow wave puts project teams in an impossible schedule squeeze.

“We don’t compromise on schedule delivery date,” noted a contractor manager, “and don’t compromise on quality, and can’t add staff, so the only variable is scope—we just kept dropping functionality. But eventually that meant we couldn’t handle all the records, and that meant we weren’t allowed to convert them [from the legacy system], and so the whole final delivery schedule got blown out of the water.” A growing mass of work had to be done at the very end—when risk was highest, and the deadline left no margin for further schedule slip.

We’re trying to put 8 pounds of slop into a 5-pound bag.

Complexity Feeds the Wave

A number of errors fed the project’s bow wave. Perhaps the most damaging one was failing to account for effects of complexity.

The team used a sequential development process, paralleling the system they were assembling: they built the initial processing modules first (handling the less complex records), and left the final modules (processing the most complex records) for last.

However, because no one really analyzed the complexity of the final modules during the planning phase, or the handling of the most complex records, the program didn’t accurately estimate the feasibility or the effort of completing these tasks.

Real-Real-Life Bow Waves

Here’s an example of bow wave behavior that many of us can identify with: a person who has trouble managing their finances, and so continually uses their credit card to maintain their standard of living instead of either reducing their spending or increasing their income. Over time they become dependent on the credit cards, but the increasing interest payments on their growing debt begin to undermine their ability to implement the appropriate long-term solutions—balancing their budget.

As you might have guessed by now, the team didn’t meet its delivery deadline and at last report was still struggling with completing the final, most complex processing module.

The Bigger Picture

The bow wave effect is an example of a dynamic where a problem is solved with a “quick fix” that gives immediate results, but only temporarily solves the original problem (see Figure 12) [Kim 1993]. The organization often knows that a more fundamental solution would be better in the long run, but feels unable to wait while such a solution is put in place. Over time the organization comes to rely on the quick fix, not realizing that it is undermining their ability to implement the fundamental long-term solution they need.

There is a pattern in evolutionary project development in which programs repeatedly fail to estimate correctly the amount of work that can be done with the resources and time available. Programs fail in this key step for many reasons, including a lack of estimation ability or historical data; the lack of a consistent development productivity rate; or deliberate underestimation of the effort needed so as to make the estimate meet the preconceived “correct” result.

```
graph TD; A((Complex Functionally Postponement)) -- "S, Delay" --> B((Design Decisions Made)); B -- "O" --> C((System Modifiability)); C -- "S" --> D((Ability to Integrate New Capability)); D -- "S" --> E((Build the Most Complete Functionality Early)); E -- "S" --> F((System Risk)); F -- "O" --> A; F -- "O" --> B; F -- "O" --> C; A -- "S" --> F; B -- "O" --> F; E -- "O" --> B; E -- "O" --> A; subgraph B_loops; A -- "S" --> B; B -- "O" --> A; E -- "O" --> B; B -- "O" --> E; end; subgraph R_loops; F -- "O" --> A; A -- "S" --> F; F -- "O" --> B; B -- "O" --> F; F -- "O" --> C; C -- "S" --> F; end;
```

Variation: Kicking the Can Down the Road

CMU/SEI-2010-TR-016 | 33

Breaking the Pattern

How can a program recognize its own bow wave? By looking at how functionality was originally allocated to releases, seeing what has been deferred, and comparing that to the anticipated complexity, maturity, or risk of that functionality. Looking at work completed versus work remaining (and checking for consistency with schedule) also can highlight a bow wave.

To break out of the bow wave dynamic, a program must first understand the cause of the original problem (schedule pressure, in our example) that leads to the expedient solution, and re-examine the other possible solutions—especially those that are fundamental. However, making the distinction between “expedient” and “fundamental” solutions isn’t always clear-cut.

Once an appropriate fundamental solution is identified, the organization must then assess its current ability to implement that fundamental solution (Is there enough time left? Do we still have the right skill set?), understanding that their ability may have eroded due to their use of the expedient solution.

5.2 Firefighting

Good Intentions

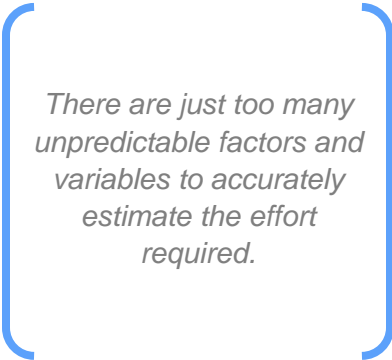
When a project begins, no one *intends* to deliver it late, or to overrun their budget, or to give users a buggy system. It just seems to happen—all too often, and despite the best of intentions.

Actually, the problems that envelop so many software acquisition efforts are predictable—which means that they are also avoidable, and often correctable.

We’re going to explore one of those predictable patterns—one called *firefighting*. A recent government development program fell prey to it after mistakes were made in the earliest estimates of the work by the contractor.

Do You Smell Smoke?

In this project, mixing the contractor’s poor estimation process with an aggressive schedule from the government yielded significant underestimation of the effort needed to develop each system release. Looming deadlines, and the probability of missing them, multiplied the schedule pressure, and work on the project became frenzied. A quality assurance (QA) analyst observed that “the



There are just too many unpredictable factors and variables to accurately estimate the effort required.

contractor burned hours like there’s no tomorrow,” yet productivity and quality fell off with increased overtime. The result: “They ended up rubberstamping code at code reviews.”

When system acceptance testing finally started, the team found the current release had a high failure rate in test cases. The government technical lead admitted the project was behind schedule “because of all kinds of bugs.”

Fire! All Hands on Deck!

The contractor’s solution? Firefighting. Pull everyone off their assigned tasks to fix the problems blazing throughout the project. Resources were pulled off of every *other* effort that was going on in parallel—notably including the next release.

Later, a team member noted that no task was safe from being stripped of people. The government acknowledged that delays on the current release would unquestionably affect the next release. The firefighting, he said, “sets my colleagues doing the next release up to fail, because then they have to start late, and their schedule will slip from the beginning.”

The contractor wanted to break out of this dynamic, but with all the people needed for estimating the next release busy fighting fires, “we’re never able to get out ahead of the problems.”

A Towering Inferno

So, the problem just got worse, and the flames hotter and higher. The contractor noted that the government deferred problem requirements—moving them to a new release—rather than facing the problem and cancelling or postponing indefinitely. The problem thus perpetuated itself, with

deferred requirements mapped to future releases, and resources diverted from early release development.

The program manager, reviewing the smoking ruin of the development plan, summed it up.

“The first-order effects of what went wrong on the release were bad enough,” he said. “It was late and over budget. But the contractor didn’t want to acknowledge that that caused the *next* release to slip, and may have reduced functionality in the current release—leaving this [mess] on the side that someone has to clean up.”

The Bigger Picture

There are many ways the firefighting dynamic can begin, but once started it is self-perpetuating. The initial trigger may be due to scope creep, budget cuts, underestimation of the actual effort, or other reasons. Processes are stressed, and shortcuts may be taken in quality processes. This allows defects to survive or be inserted into the system.

The contractor burned hours like there’s no tomorrow.

Reading the Causal Loop Diagram

A program has a desired goal for the number of allowable defects in the delivered system—and the difference between that goal and the actual number of problems is the *Problem Gap* (see Figure 13). If this gap increases, then *Resources Dedicated to Current Release* must increase to do rework to fix problems.

How can I break this vicious cycle of schedule skips, cost overruns, and high defect rates?

More resources doing rework means fewer *Design Problems in Current Release*, and reduces the *Problem Gap*. This is a *balancing* loop in which rework offsets (balances) the defects being inserted. Unless the staff size increases, more people assigned as *Resources Dedicated to Current Release* leaves fewer *Resources Dedicated to Next Release*. This reduces the resources available for *Early Development Activities on Next Release*—which, after a delay, increases the number of *Design Problems in Current Release*.

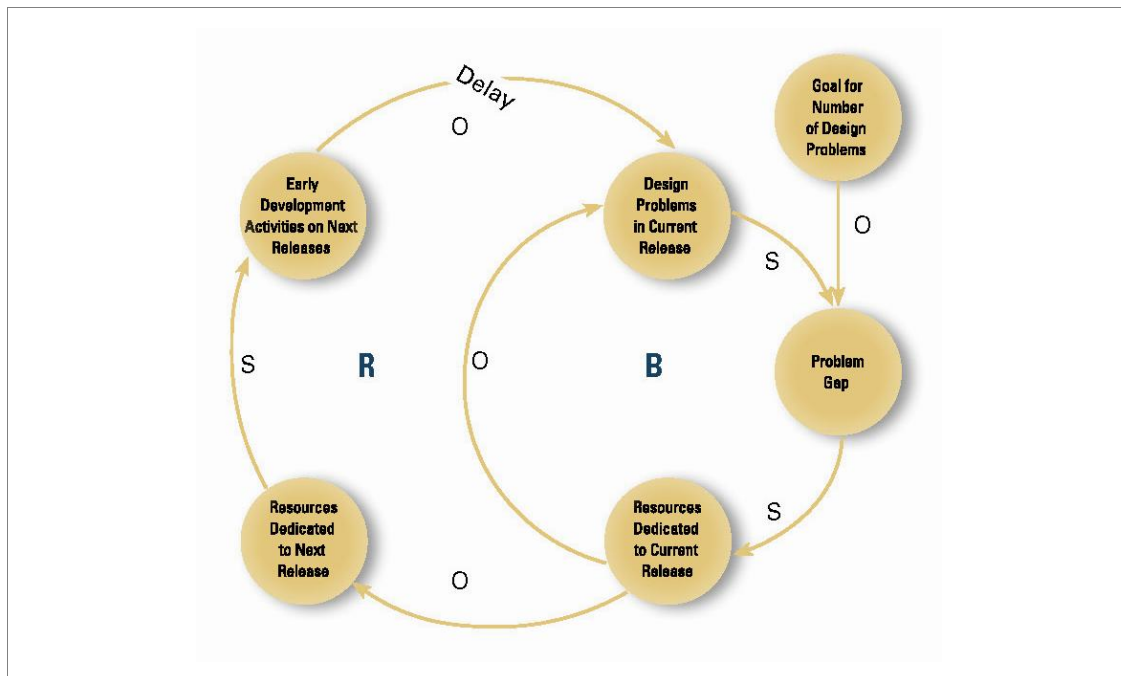


Figure 13: Causal Loop Diagram of "Firefighting"

This exemplifies the classic problem of trading off long-term benefits for short-term gains, and results in exacerbating problems rather than resolving them. As a result, additional resources will have to be spent in resolving the new problems introduced into the future releases.

Breaking the Pattern

From a systems thinking perspective, to break out of this ongoing dynamic this program needs to: (1) acknowledge up front that the "fix" they are using—namely diverting resources to address problems in the current release—is just alleviating a *symptom* of the true problem, and (2) commit to solving the *real* problem—accurately estimating the time and effort required for a release, and staffing each new release in accordance with that estimate from the beginning so that more problems with quality don't occur [Kim 1993].

For other programs that have not yet experienced this type of behavior, there are ways to avoid it [Repenning 2001]:

- Don't invest in new tools and processes if you're already resource-constrained.
- Aggregation of resource planning (across all subtasks) is critical to fire prevention.
- When a project does experience trouble in the later phases of the development cycle, don't try to "catch up"—revisit the project plan instead.
- Don't reward developers for being good firefighters.

5.3 Everything for Everybody

Measure Twice, Build Once

A program sponsored by several services was trying to build a software infrastructure for communications that could be used on platforms for air, sea, and ground vehicles. A common system offers significant cost savings over custom software for each different platform.

They tried to be too many things to too many people.

Before the contract award, five platform programs agreed to use and fund the infrastructure software, placing it on the critical path of their schedules. It was important for the program to have platforms committed to using the software and contributing to fund the development—that demonstrated need and helped defray costs. That

commitment by the initial five platform programs generated the interest of still more programs—and necessitated discussions about new infrastructure requirements needed to support these additional platforms.

Too Much of a Good Thing?

Commonality is a great objective, but sometimes there can be too much of a good thing. In this program, it meant that the infrastructure software was going to have to deal with multiple platforms with varying requirements—a capability that would come at a steep price in terms of cost, effort, and complexity. As one engineer on the program observed, the system “has to be complex to do this job. It could only be simpler if the requirements were fewer, or simpler.”

The program “needed platforms to get funding, but that means taking on differing requirements,” said one program lead. “So it has to be configurable, which brings in software complexity.” Complexity translated to additional development time and effort.

The number of platforms that needed to use the software to sufficiently amortize the cost meant that more custom requirements had to be addressed and resolved. “We wanted to involve [the platform programs] as early as possible, so they could cooperate,” one manager noted. “That keeps them involved, but it allows them to drive design, and push us off track.”

Running Out of Time

In order to keep the platform programs committed to using the infrastructure software, the team had to rush to meet the “need dates” for the various platforms. This forced the program into an aggressive, 18-month schedule—a schedule in which everything would have to go like clockwork. It was clear, said one team member later, that the infrastructure program “was trying to do too much in too little time.” A key program management review with the contractor held the next year showed that they were falling behind.

Jumping Ship

The program lost more momentum as tight budgets and funding cuts forced two intended platform programs to drop their plans for the new infrastructure. Not long after that, a third, key platform

program decided the cost growth was too much, and with it, one of the participating services then backed out of the infrastructure program entirely, and all of its platform programs went with it.

Cost wasn't the only issue, according to one manager. "In most cases platform programs have pulled out due to schedule slips," he said. "[The infrastructure program] can't deliver the capability required in the timeframe the program has to have it."

As the number of platforms declined, new requests still came in to the infrastructure contractor from the remaining platforms, whose happiness had now become a critical concern. As one frustrated team lead put it, "Every time [the platform customer] said we must do this to make it work, we rolled over and agreed to do it."

You Can't Please Everyone

The program continued on, unable to amortize its costs across just the remaining platforms, a year behind its original 18-month schedule, and unable to justify the additional development needed to support more platforms. "This would have been a totally different program if we didn't need to build a general-purpose solution," observed one member of the program staff.

In the words of a program official, "[they] bit off more than they could chew, trying to do everything for everybody. You sacrifice too much, making it too complex. If you had scaled it down a bit it could have been done faster, and more easily."

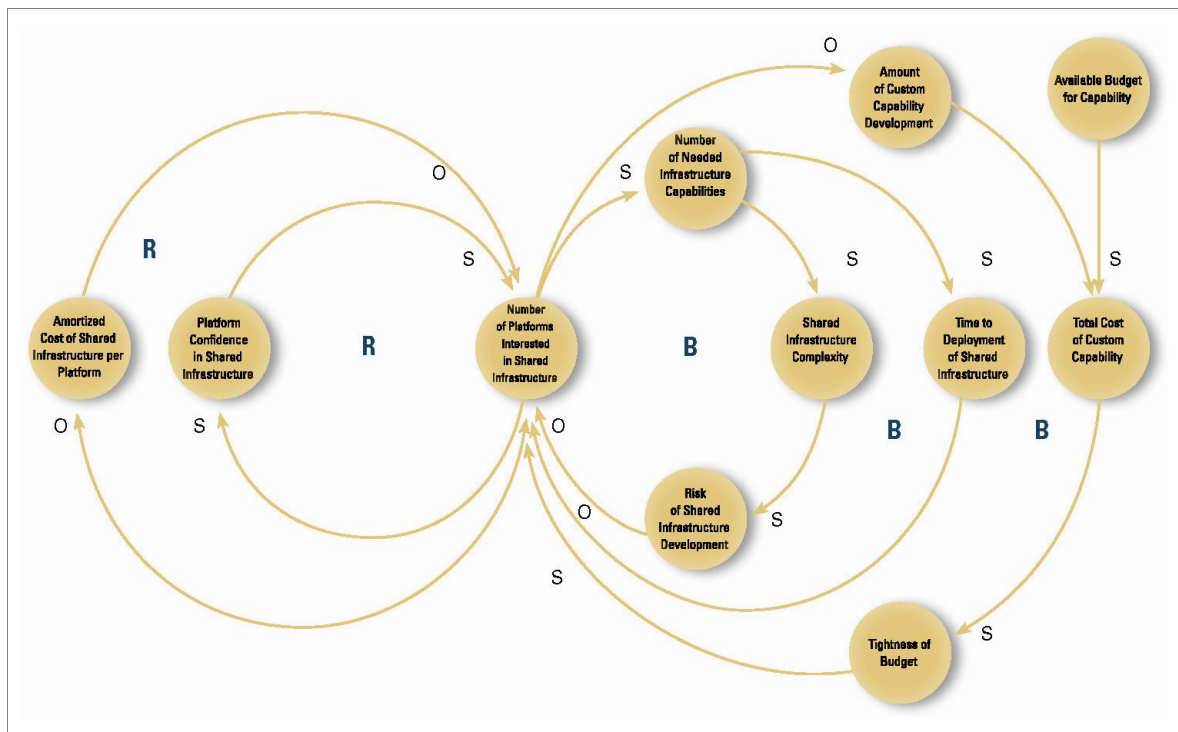


Figure 14: Causal Loop Diagram of "Everything for Everybody"

The Bigger Picture

This acquisition archetype is an instance of the system archetype “Limits to Growth,” in which initially rapid growth slows because of an inherent capacity limit in the system that worsens and increasingly undermines growth as more growth occurs.

In our example, the infrastructure program attempted to reconcile the competing requirements of the different platforms by creating a single software system that would fulfill all of the different platform requirements. Furthermore, the platform programs made their aggressive delivery schedules a requirement for success. When the infrastructure program slipped and cost grew, the platform programs opted out. This loss of funding resulted in further cost growth for the remaining platforms, in turn causing more programs to back out of the program.

This archetype presents a reinforcing loop of increasing platform interest as more platforms sign on and the cost per platform steadily drops. However, this growth then potentially reverses by a balancing loop that represents the side effects of increasing complexity, cost, and delivery schedule from the number of participating platforms, that undermines and erodes platform interest.

Breaking the Pattern

Once this dynamic starts, it is difficult to stop. Prevention is clearly the best remedy.

Platforms that have schedules too short for the infrastructure program to realistically meet should not be considered as viable candidate participants. Furthermore, the program office should act to “hold the line”—to avoid letting the attraction of more funding and support force unwise decisions regarding the number of capabilities that can be delivered by a single system. This can be done by evolving the set of infrastructure capabilities slowly and modestly based on return on investment—starting with the smallest set of capabilities that will provide the highest value to a small set of platforms.

One other option that can minimize this dynamic is to either provide incentives for (or even mandate) the use of the common infrastructure by individual platform programs. Mandates, however, may be unpopular because they impose an external dependency on a program over which the program office has no control—and can thus become a risk. Incentives can take the form of economic advantages offered to the platform program to balance the additional potential risk the program takes on by choosing to use the infrastructure.

Short of an incentive or mandate, potential platform programs could be required to do a cost-benefit analysis between using the infrastructure solution and developing a custom solution. This approach at least ensures that the program is aware of the cost savings that are possible by using the infrastructure, and that the risks of both approaches are weighed.

After the infrastructure has been successfully delivered and integrated at least once, many of these issues become moot. Credible numbers for integration cost and effort should now exist, as well as data on performance of the system in the field—both addressing many of the risks that otherwise might affect potential platforms.

5.4 Feeding the Sacred Cow

Toil and Trouble

Most programs gain momentum as time passes. Some, though, take on a life of their own—after a number of milestones pass, and teams expend time and effort, they seem self-propelled, unstoppable. They’re woven into an organization’s existence. They seem privileged, too, despite the fact they often have yet to show any value to customers or stakeholders. These few “precious” programs become sacred cows: they are fed, protected from harm, and are often revered. They are beyond reproach.

Sometimes ... well-established programs ... are not tolerant of even healthy criticism.

But all systems also have issues—and it is healthy to raise risks and problems with a system in development (especially major ones). That way all concerned become aware of the issues, and can commit to finding the best possible resolutions. Unfortunately, sacred cow projects are neither subject to nor tolerant of even healthy criticism and dissent. In one real-life example, a program was fielding an IT business system to a network of field offices. The

project was several years into its timeline and nearing its initial beta test deployment, yet had long before become largely off-limits to any active questioning by the organization.

Problems? What Problems?

For example, even as stakeholders attempted to raise issues during development, their questions—and their acts of criticism—were rebuffed by project staff and managers. The project team became increasingly defensive:

- User concerns about creating a centralized system architecture with a single point of failure were dismissed.
- Disagreements with choosing the second busiest site in the nation as a beta site were ignored.
- Concerns over rushing to a cut-over date before the system was ready were downplayed.
- Criticism from whistle blowers, the media, and Congress about serious issues after initial roll-out began elicited only defensiveness.
- Program team members characterized disgruntled users as incompetent or computer illiterate—warning other critics to back off.

This cow could moo.

Hey! Keep Feeding Me!

As these and other risks arose, the program office and the contractor repeatedly deflected them. In continuing to receive funding—and continuing to throw money at the project—they blamed others, or shrugged the problems aside as irrelevant. Any questioning of or disagreement with the program’s direction or approach met unresponsiveness or hostility. This single-minded support of the program, even at the expense of the stakeholders’ interests, marked a form of defensiveness by the PMO and contractor. It affected the objectivity of the decisions being made, and the program

proceeded on its obdurate path. Decisions believed by many to be fatally flawed went unchallenged, yielding only further development investment to implement those decisions.

It's the User's Fault

Six months after “go live” became a disaster, the contractor still denied there were any significant technical problems with the system—that it was entirely a case of user incompetence. Of course, the sacred cow still was fed—substantial time and effort continued to be invested in system development.

Six months after ‘go live’ became a disaster, the contractor still denied there were significant technical problems.

The Bigger Picture

The general phenomenon of *escalation in decision making*—of which this archetype is an example— is widely recognized, and described as “persistence with a venture beyond an economically defensible point” [Drummond 1996]. Decision making on large, high-visibility programs becomes less technically objective and more politically defensive as time passes. Various factors may come into play in producing the effect, including uncertainty regarding the outcome of the program, poor visibility into program progress and status, sunk cost, prior decisions, personal self-interest, and ego- and face-saving.

As the Figure 14 illustrates, various *System Issues* continue to arise during development. These are dealt with through a series of *Effective System Investments*, maintaining equilibrium within a balancing loop. However, increasing *System Issues* produce *Criticism of [the] System*, which then drives up the level of *Personal Investment [and] Defensiveness*, reducing *Objectivity* on the part of the decision-makers and reducing the *Quality of Investment Decisions*. The lower *Quality of Investment Decisions* in turn reduces the value of the *Effective System Investment*. This creates a reinforcing loop that surrounds and can ultimately overwhelm the original balancing loop by increasing *System Issues* in a continuing cycle.

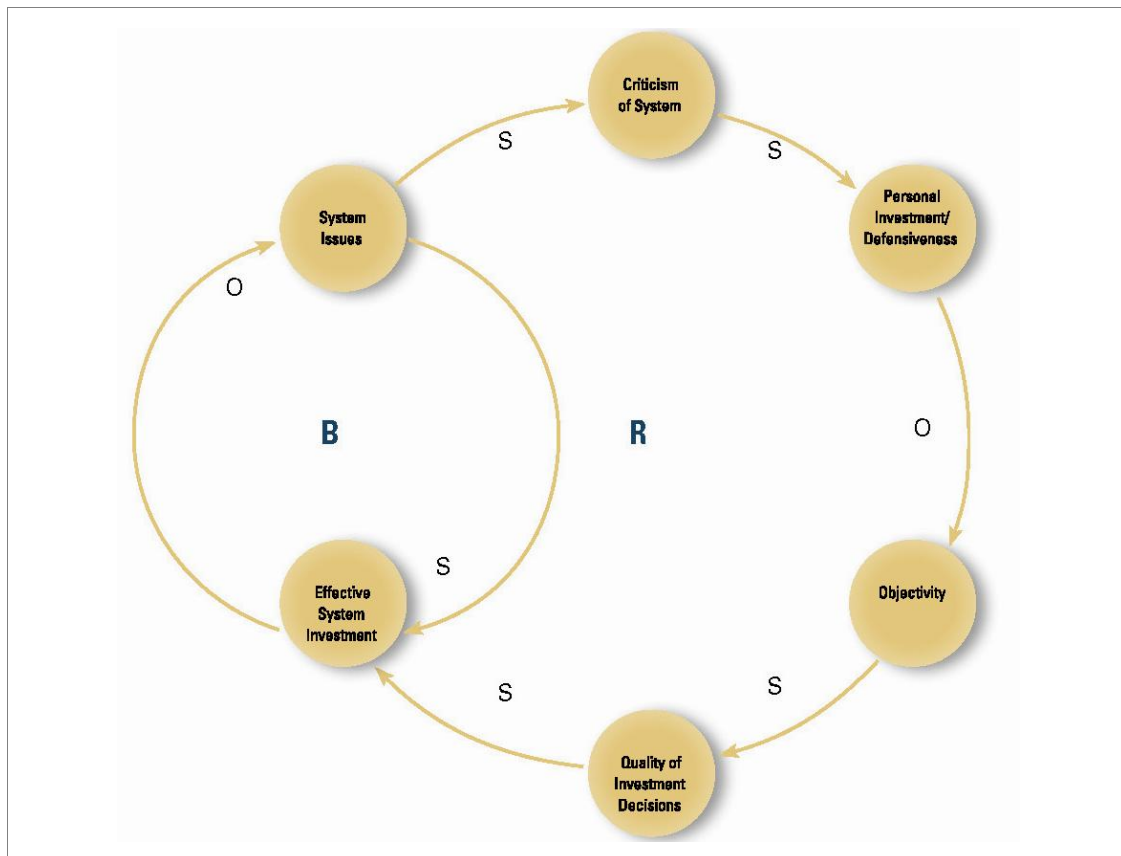


Figure 15: Causal Loop Diagram of “Feeding the Sacred Cow”

A key aspect of this dynamic is the loss of objectivity on the part of the decision makers. They have become too close to the project to be impartial, making them unable to assess the true feasibility of the system. It may be due to ego or stubbornness [Flowers 1996]. Regardless, the results are likely to include overly optimistic status reports and a “desire to commit more resources to improve things.”

Breaking the Pattern

Recovering from “Feeding the Sacred Cow” requires recognition that the counterproductive behavior is taking place—recognition by the very people who are embedded in the dynamic and have lost the ability to make objective, rational program decisions. If “Feeding the Sacred Cow” has taken hold of the program, a significant change in management may be necessary to “reset” personal factors such as self-interest, ego, and face-saving.

Another key step in recovery is to conduct a series of formal, objective reviews with external technical experts to identify and address issues. This on-going review process will bring the original program goals and assumptions back into focus, test them for continued feasibility, and help decision makers make rational choices.

To help prevent this escalation behavior from taking hold, several steps should be taken:

- Actively encourage dissenting opinions; don't shoot the messenger. Honest, objective resistance to the program can help solve problems early, when the chance of resolution is greatest. To leverage dissent, establish a formal process to raise, review, negotiate, and resolve issues in a way that stakeholders can agree is fair.
- Let technical rationality rather than political considerations guide decision making. Planning regular, technical program reviews (such as the one described earlier) is one good step to take.
- At a minimum, regularly review and question the original assumptions behind the decision to develop the system. Are they still true? This is a great preventive measure. Determine if it's still possible to move forward, if a change in direction is needed, or if the original rationale has changed such that the program is no longer relevant.

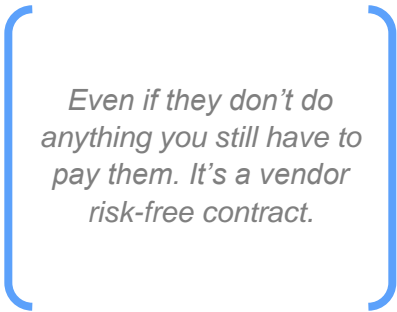
5.5 PMO Versus Contractor Hostility

Relationship 101

A good relationship between the government program management office (PMO) and the prime contractor is the foundation of a successful acquisition, in the same way that a trusting relationship is the foundation of a good marriage. Of course, the reverse is true, too: any seeds of distrust planted at the beginning of a relationship, if nurtured, can destroy it. In business as in marriage, credibility is lost. The presumption of innocence is replaced by an assumption of malice. Both parties go on the defensive, determined not to be taken advantage of. The stage is set for drama and disappointment.

Newlywed Squabbles

Looking back on one recent example, a program official said the PMO and contractor relationship started “with high hopes, and the best of intentions on all sides.” The contract was structured as “sole source and a 15-year marriage.”



Even if they don't do anything you still have to pay them. It's a vendor risk-free contract.

As the development began, the difficulty of the work surpassed what had been expected and planned for. In fact, the amount of functionality in each release fell short of the government's expectations. The PMO became skeptical of the contractor's ability to meet deadlines, even as the contractor was pointing out a need for “slack in the schedule for managing the risk associated with this development.”

Although dissatisfied, the PMO realized little could be done. This was a sole-source contract, one government official said, “the government has no leverage. It [sole source] removes the motivation to be a sincere partner.” Another official concurred. “Even if they don't do anything, you still have to pay them,” he said. “It's a vendor risk-free contract.”

Pointing Fingers

The ability of the contractor development team came under fire. One high-level government program official said, “It's not a marriage—it's not even a partnership. We're not getting the best engineers, the best managers, or the best development team.” Once the government had concluded that the contractor was unreliable, its managers felt their only option was to “tighten up on them” and “hold their feet to the fire.” As a result, even when the government saw early on that a schedule slip was inevitable due to delays in preliminary design, the PMO team deliberately didn't perform any risk mitigation or contingency planning: “they wouldn't let the contractor off the hook.” The government's strategy to force the contractor to perform acceptably was now not just extending the conflict—it was actually worsening system cost and schedule performance.

Another point of contention was contractor access to government subject matter experts, or SMEs. The contractor received poor documentation of the legacy systems it was trying to replace—so the contractor asked for access to government SMEs. However, the government had only assigned a

small number of SMEs to the program, and they began to push back, saying that it was “the contractor’s job to figure all of this out.”

The result of all the conflict between the parties—schedule problems, perceived capability inadequacies, unwillingness to provide SME assistance—led to general mistrust by each side for the other. They traded disparagements, and the bad feelings escalated.

It’s not a marriage—it’s not even a partnership.

The government asked for too much capability in each release, the contractor complained. The government has no confidence in the contractor’s estimates, the PMO countered.

Heading for Divorce Court

As the relationship deteriorated, hostilities escalated. The government felt that the contract was “a recipe to milk a cash cow forever,” and acknowledged that it would like to end the relationship. However, a top government executive admitted “being beholden to the contractor...because if the contractor chooses to walk, or if the government says, ‘You’re banished,’ I don’t know what we’ll do.”

Results

Ultimately the PMO became resigned to consistently late releases, and what it believed were inflated estimates for requested work. In turn, the contractor was forced to provide even more heavily padded estimates. These protected it from the government demanding more than could be provided in each successive release.

The Bigger Picture

In the “PMO versus Contractor Hostility” archetype two parties destroy their relationship through tit-for-tat retaliations for actions they *perceive* as being harmful to their interests. While they start out with the same goals and the best of intentions, at some point one partner takes an action that is in their own best interests, but is harmful to the other. When the other partner views that action as deliberate and a surprise (and perhaps hostile), it responds with an act that protects itself from the initial act. This response also may “send a message.” However, it may also, in turn, surprise and anger the first side. After a series of such actions the two sides can become sworn enemies rather than the intended cooperative partners. Only the smallest perturbation is needed to push this dynamic out of its equilibrium and start it sliding into hostility.

The irony in the example story is that the actions the government takes to deal with the contractor’s perceived “bloated estimates” become a self-fulfilling prophecy, creating the very inflated estimates they were intended to prevent. The contractor has little choice other than to pad the estimates further to help ensure that they can be met in the future.

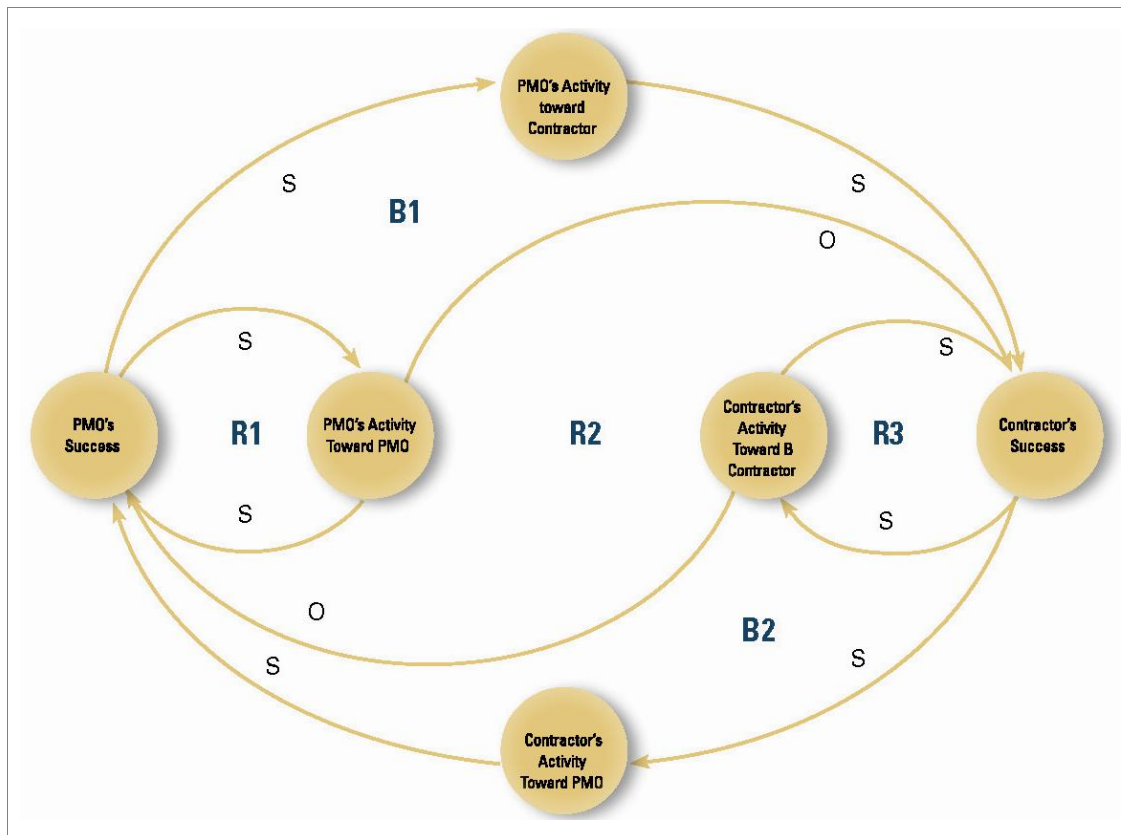


Figure 16: Causal Loop Diagram of “PMO vs. Contractor Hostility”

Some innocent acts the contractor might perform that the PMO could misinterpret as deliberately provocative include the following:

- Missing delivery deadlines as the result of trying to be too accommodating
- Hesitating to accept a small proposed modification to a system requirement because, regardless of size, it is a modification that falls outside of the contractual agreement, and thus needs an engineering change order

Some examples of acts the PMO performs innocently that a contractor could misconstrue as punitive or unwarranted include the following:

- Withholding a subjectively evaluated award fee
- Providing system requirements that haven't been thought through or precisely expressed, obligating the contractor to do additional clarification on the requirements (and then making the contractor fully accountable for the resultant missed deadlines)

Breaking the Pattern

To stop the dynamic, first the cycle of escalation must be broken, and then, in cases where trust is lost, both sides need to signal their commitment to restoring it. This formal signal of commitment must have a substantial cost associated with breaking it: loss of public image, financial value, or something similarly valuable.

The signal must be a significant, unilateral offer that is initially extended by one party.

This formal commitment is necessary because, with both parties enmeshed in the dynamic, it is not sufficient to simply start living up to the original expectations of the relationship. Both parties must now work *harder* than they would have had to at the beginning to re-establish the trust that has been lost.

Pre-Marital Counseling

Of course, the best way to deal with counterproductive behavior is to prevent it from ever starting. Assure that there is a healthy PMO and contractor relationship *before* the real work begins—rather than going through a painful reconciliation after the marriage. It is true that the PMO has a vital oversight role with respect to the contractor, but that needs to translate to a policy of “trust—but verify,” with the trust clearly demonstrated.

Finally, establishing and keeping open lines of communication between the PMO and contractor will not alone prevent or end hostility. But without communications, no other actions will succeed. Both parties need to have a method and opportunity to easily talk to one another about *what* actions each party is taking, as well as *why*.

5.6 Staff Burnout and Turnover

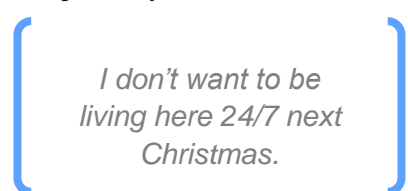
Applying more pressure on staff can temporarily increase employees' productivity, but burnout soon sets in. This results in lower productivity, slowed progress, and even greater schedule pressure than before.

Pressure!

In our sample case, the program had been active for some time attempting to update an agency's IT systems and infrastructure. As one program executive put it, "There's a lot of pressure on us since agency modernization has been going on for quite a while. The program is seen as the foundation project for modernization. If it backslides, it splashes on everything else. We have to meet these milestones, or else the agency modernization program will be seen as failing."

Burning Hot ... and Burning Out

The contractor felt the pressure to deliver, and responded by working harder. One development manager admitted: "My people are working overtime right now. I am here every day from 9 to 9, except Friday, and more than half the team was here Saturday and Sunday." One developer



*I don't want to be
living here 24/7 next
Christmas.*

complained, "I don't want to be living here 24/7 *next* Christmas." The government program manager was aware of the long hours being put in by the contractor, but was not entirely sympathetic, saying "they're always 'burning hot' because they're always late."

Quality Takes a Hit

The immediate casualties of long hours were quality and productivity. These problems might have been caught and corrected under normal circumstances, but as deadlines mounted, "Code reviews and unit test reviews [were]...not maintained...because of the growing schedule pressure," one team member explained. When errors crept through, quality suffered, but when they were caught, they had to be fixed. This consumed more time—time they couldn't afford.

The longer term effects were perhaps even more dire. One contractor manager pointed out that with the long hours and declining morale "...the risk of burnout [became] an issue."

Let Me Out of Here

The government began to see the consequences of the ongoing high pressure, with program office team members admitting that "They've had a hell of a turnover over there" (on the contractor's development team). The turnover began to synchronize with the release cycles as the stress levels ratcheted up.

Hiring Replacements

The loss of experienced developers exacerbated the program's plight because of the difficulty of replacing them. In the words of one technical manager, "You can always replace bodies, but it's hard to lose critical experience. I think that only a handful of people are left here, with experience, since two years ago."

No Way Out?

Government and contractors can now see how damaging a pressurized project environment can be, as Brooks' Law catches up with the program—bringing on new people becomes the primary need, but hiring is expensive and time-consuming.

The government believed the contractor should have prepared for staff changes, with one top manager saying, “[The contractor] should have junior programmers that they’re bringing up to speed, but they haven’t done that.”

We have a 61 percent attrition rate—that’s a huge, core problem.

Now that the cycle has taken hold, is there a way out?

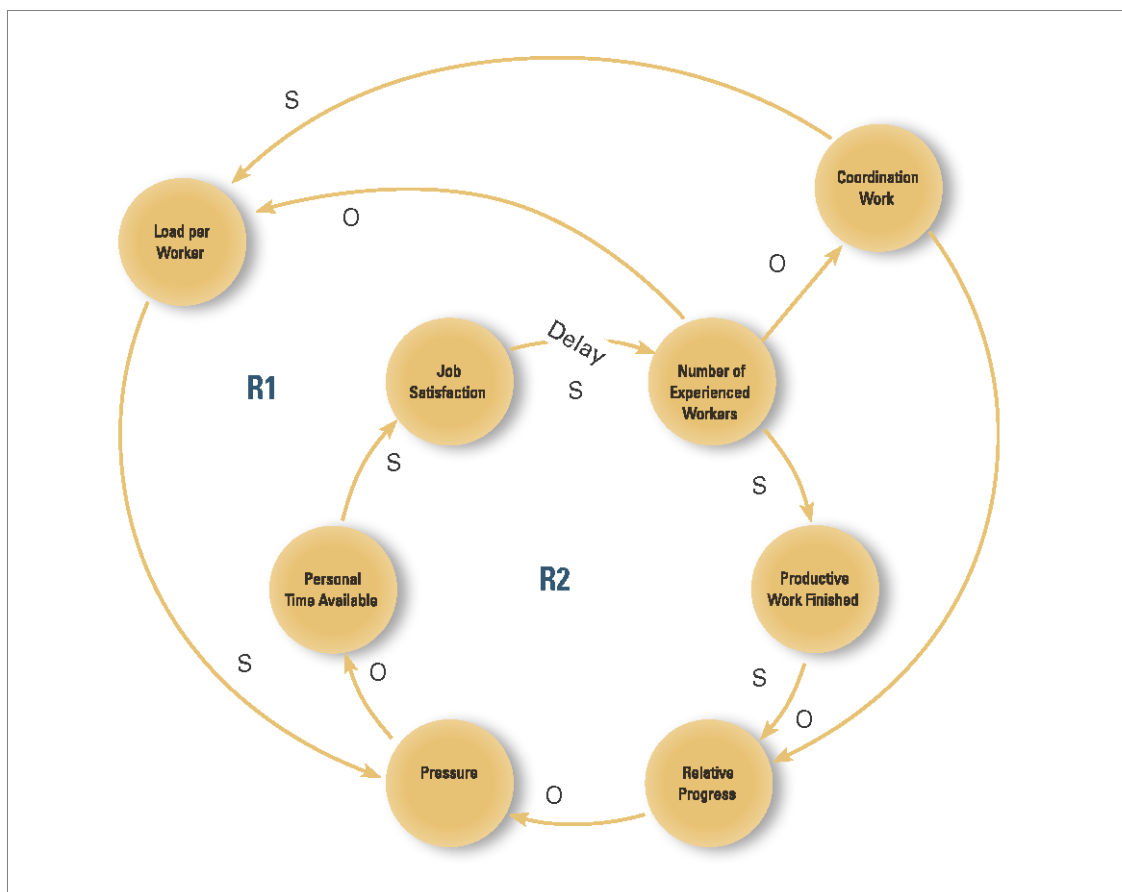


Figure 17: Causal Loop Diagram of “Staff Burnout and Turnover”

The Bigger Picture

Abdel-Hamid discusses the pervasive effects of pressure on a development team in *Software Project Dynamics*:

Consider[ing] the impact of schedule pressure on the workforce turnover rate.... There is evidence to suggest that workforce turnover increases when scheduling pressures persist in an organization. This can be costly, since a higher turnover rate translates into lower productivity on the project [Abdel-Hamid 1991].

Turnover is the direct result of poor job satisfaction. Employees are unsatisfied when there is a significant gap between the work environment they *want* and the work environment they *have*. When work conditions become sufficiently egregious, the employee must either improve their situation in the organization, or move to another organization. The latter is turnover. There are several different effects going on simultaneously in this archetype:

1. Continuing pressure is driving down morale and *Job Satisfaction*, leading to burnout and turnover.
2. The damage resulting from experienced workers piles up:
 - Progress is reduced (primary effect).
 - *Coordination Work* is increasing (secondary effect).
 - Workload/pressure on remaining staff is increasing.

Breaking the Pattern

Staff productivity maintains an equilibrium. Sustained (or increasing) pressure destabilizes that equilibrium, starting a downward spiral of burnout and turnover. When such schedule pressure begins, the program must find alternative ways of relieving that pressure to maintain stability. If a program is under constant and inordinate schedule pressure and the situation is allowed to continue, the net effect will be to burn out the staff, see them leave, and then watch the program collapse under a negative reinforcing loop of turnover.

The choices to break this pattern are to: (1) reduce the scope of the project, (2) slip the schedule, or (3) add manpower.

This last option lands the program squarely back in Brooks' Law territory—adding manpower to a late software project—and has the same consequences [Brooks 1995].

Prevention of the *Staff Burnout and Turnover* dynamic is more desirable. This approach requires two vital elements:

- The PM must find another solution to the problem. Passing sustained schedule pressure on to the staff quickly becomes unproductive and then counterproductive.
- Be willing to invest in a quality work environment to keep your experienced people on the team. Doing this will be far less expensive in the long term than replacing them.

5.7 Underbidding the Contract

Bidding on a Contract

The concept of bidding on a government contract is deceptively simple. A contractor is looking for work. The government has a job that needs to be done, and issues a request for proposal (RFP) describing what they want. The contractor estimates what it will cost to perform the work, and submits a proposal, which includes a bid. If the government accepts the proposal and the price tag, the contractor wins the contract.

Increasing Your Chances

Bidding on a contract is serious business, with costs anywhere between \$1 million and \$5 million to bid on a \$50 million or \$100 million contract. A contractor needs to know everything possible about the program, and have high confidence that its bid will win.

Bad programs are good business—at least for those willing to work that way.

Because government contracting is competitive, each contractor looks for ways to make its bid stand out from others as more attractive and thus increase its likelihood of winning. One approach is to *underbid* the contract—that is, bid *less* than the amount the contract will actually cost to perform. To do this, the contractor must find out how much money the government has planned to spend on the work.

This is sometimes accomplished through personal networking. In our example case, one PMO staffer said, “The retired acquisition program manager, who is now with the contractor, can call his buddy at the acquisition program, and find out the program duration and available funding.”

Also, contractors have access to descriptive summaries and can get a feel for the overall program; and they may know the value of related contracts. Budget information, including funding requirements and profiles, is also often included in the RFP.

Often the contractor collects enough information about the program that they can decide in advance whether to bid for it.

Making an Underbid Pay Off

When a program is underbid and won, regardless of the intent, the program now has inadequate funding to complete the planned work. Naturally, this leads to shortened schedules or understaffing, which may cause schedule slips or pressure, and quality shortfalls. To pay for these, the contractor will want to find a way to “recover” the money that was “lost” from the underbid. This can be accomplished in various ways.

With cost-plus contracts, a contractor may be able to make the money up on the award fees and incentive fees. In a cost-plus contract environment, a schedule slip is tantamount to receiving additional funding. Alternatively, the use of engineering change proposals (ECPs) (work not included in the original contract) that feed off requirements scope creep can direct extra incremental revenue to the contractor.

Another approach may be to make back the money lost on the development contract in the *production* contract—where a large portion of the funding resides. The government may be unhappy with these actions, but unless it is willing to expend great effort it is largely locked into continuing to work with the contractor to complete the contract.

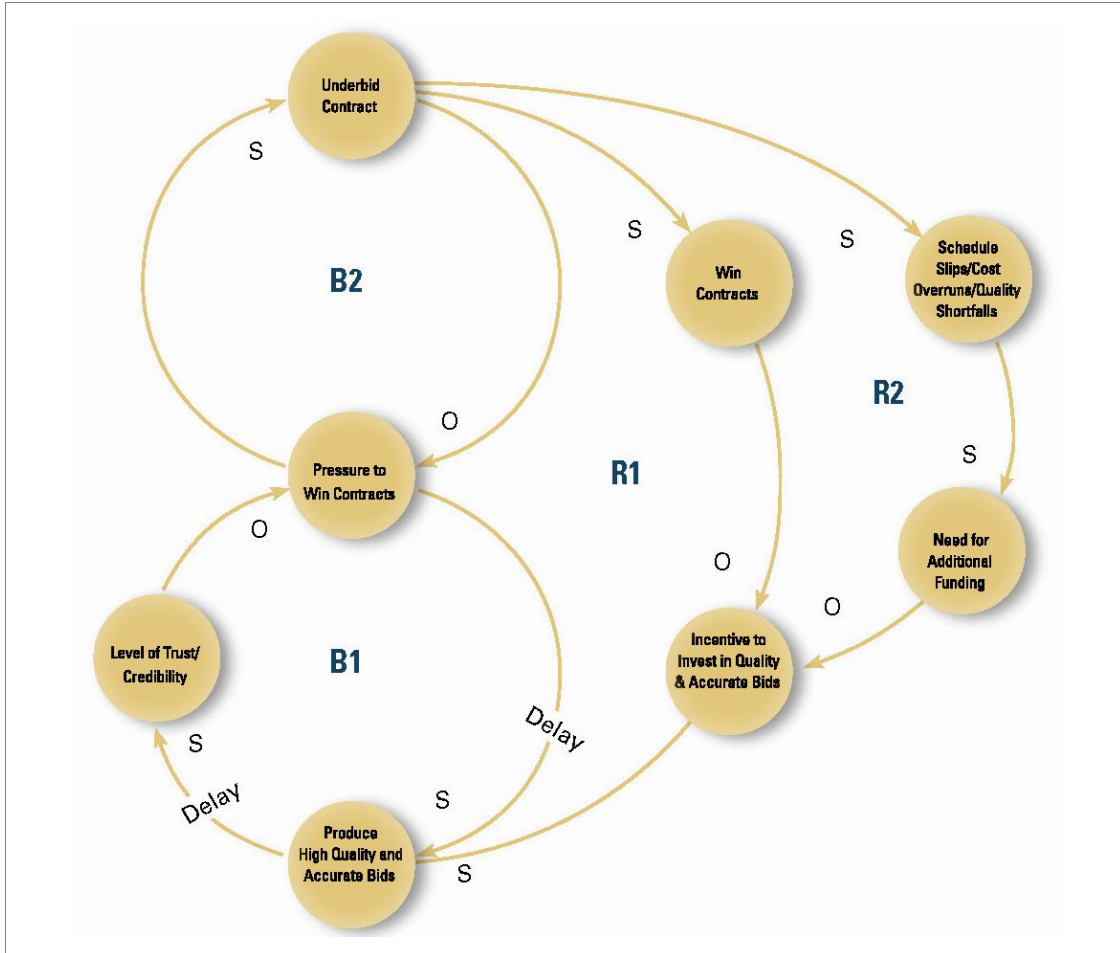


Figure 18: Causal Loop Diagram of "Underbidding the Contract"

The Bigger Picture

Underbidding the Contract is an archetype whose behavior may occur across multiple programs. The use of this strategy evolves over time, and a reinforcing behavior sets in that increases the likelihood of underbidding. Contractors who underbid find that they can both win contracts and make the underbids actually work: when cost, schedule, and quality problems emerge later, the contractor receives additional funding and schedule relief to allow it to complete the job. This encourages other contractors to underbid the *next* program they compete for. This pattern can result in negative outcomes, such as confrontation between the government program office and the contractor. However, the money that the contractor generates may be enough to compensate. In short, this may be a viable (if

The contractor will want to find a way to "recover" the money that was "lost" from the underbid.

fundamentally flawed) business model—“bad programs are good business,” at least for firms willing to work that way. If underbidding is allowed to flourish, some competitors lose incentive to produce accurate bids, because by doing so they will not win contracts, and may ultimately go out of business.

It is difficult to attack underbidding by tightening ECPs. ECPs are commonplace due to changing environmental and technological factors, and aren’t likely to be viewed with suspicion, since the technology will advance and offer new potential capabilities that were previously unimagined. Stakeholders learn more about what the system as specified *will* do, versus what it *could* do—and invariably want it to do more.

The motivation underlying this archetype is varied. From an innocent perspective, if the program’s complexity is underestimated, then the cost and schedule will likely be underestimated, as there will be unforeseen technical problems. However, this doesn’t explain why underbidding and its attendant issues occur so frequently in acquisition programs—an observation which points to underbidding as an intentional response to the acquisition contracting process.

Breaking the Pattern

Breaking this pattern completely is not the responsibility of a single PMO, nor could it be solved by a single PMO. However, the PMO still needs to take action to try to prevent it from occurring, because the downstream effects of underbidding on their program will still be highly damaging.

To minimize the likelihood of an underbid, the PMO needs to do the following:

- Make bid price a lower priority consideration compared to the total value offered by the contractor’s proposal.
- Provide comprehensive technical detail in the RFP and conduct a thorough technical evaluation of the proposals to ensure that the contractor has a detailed understanding of the effort involved.
- Double check the given estimate against the work proposed.
- Be suspicious of a low bid during source selection based on the bid price compared to the independent government estimate (although it may be difficult to confirm until development).

If the PMO determines a substantial underbid has likely been made, the PMO needs to act to establish a new, more accurate baseline cost estimate, communicate this new reality to executive management, and choose a way to proceed. The options here can range from restructuring the contract (from the incentives to the production contracts) to terminating it altogether, and may depend in part on the degree of culpability that the PMO assigns to the contractor.

5.8 Longer Begets Bigger

Background

In 1983 a military helicopter program was started to develop an advanced aircraft for performing armed reconnaissance in all weather conditions. The new helicopter would also incorporate stealth technology.

The acquisition included a nine-year demonstration/validation (DEM/VAL) phase before beginning an engineering and manufacturing development (EMD) phase to build the production helicopters.

Although launched in 1983, the program did not plan to deliver production units until 2006—an expected acquisition and development period of *23 years*.

Budget Cuts, Slow Development

The acquisition approach to the helicopter changed substantially during the long course of the program. Over its lifetime the program was restructured six times due to budget cuts. After one severe reduction, a major schedule extension was made to allow development to continue, but at a very low funding level, which further slowed the pace of the development.

A decision was made 15 years into the program to accelerate development of some of the helicopter's critical subsystems, but to do so within the existing funding. This accelerated development required instituting a significant number of new acquisition processes on the contractor team, adding to the program's overall risk.

Manufacture Under Scrutiny

Completion of the long DEM/VAL phase was followed by a successful milestone review of the program's readiness in 2000—and, with it, approval for EMD.

The per-unit cost had more than quadrupled since initial development.

Yet the program came under increasing scrutiny as development continued. This was in part because of its high total cost estimate of \$38 billion—\$14 billion of which was to be spent between 2004 to 2011, with much of that allocated to manufacturing. An early plan envisaged procurement of 5,023 helicopters. However,

the per-unit cost had more than quadrupled since initial development, causing the military to incrementally slash its planned production quantities down to 1,400, then 1,213 and finally to only 650—less than one-eighth of the quantity originally envisioned.

Cancellation

Ultimately, the program was cancelled in 2004 after 21 years, \$8.5 billion dollars spent, the construction of two flying prototypes, and a partially completed test program. The helicopter was still at least two years short of going into full production. The reasons for the program cancellation included the need to invest in renovating the existing fleet of aging helicopters—

which had become even more important in light of the past postponements in delivery of the replacement aircraft.

Also, the world situation and intended operational environment for the helicopter had changed substantially since the program's inception. As military threats changed from the Cold War era to counter-terrorism, the corresponding changes that would be needed to make the helicopter survivable would have added several more billion dollars to the total price and affected its stealth performance.

Meanwhile, a new technological alternative, unmanned aerial vehicles (UAVs), was coming into use in the surveillance role at lower cost, at no risk to the warfighter. UAVs had already proven their worth.

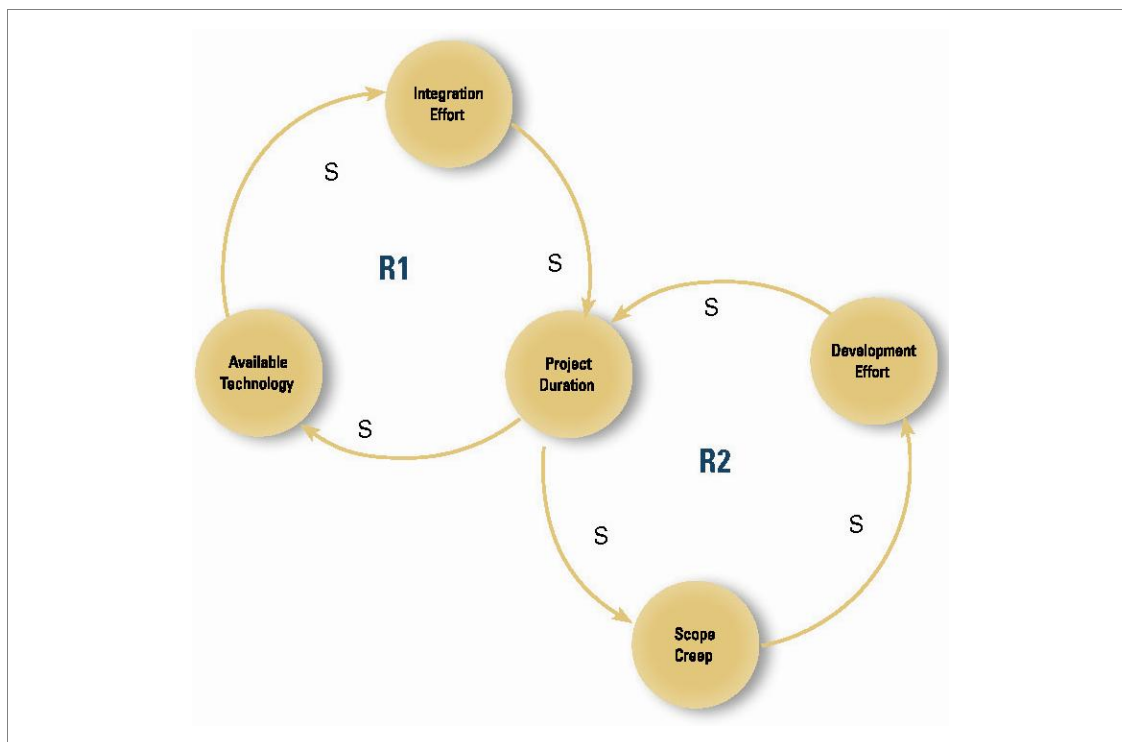


Figure 19: Causal Loop Diagram of "Longer Begets Bigger"

The Big Picture

Establishing a long development period in the initial plans actually contributes directly to expanding costs and schedules—what was expected to take a long time ends up taking even more time. This occurs for two reasons: (1) longer project duration leads to greater project effort, and (2) greater project effort leads to longer project duration.

Longer duration leads to greater effort because of steady environmental changes and ongoing scope creep. Greater project effort leads to longer project duration simply because additional effort requires additional time to execute.

In this dynamic several things can happen. The technology can become obsolete before it is time to field the system (thus forcing a redesign). The user or operational needs may evolve past what the system was designed to do by the time the system is delivered, rendering the delivered product inadequate or irrelevant. (That, in turn, can force either a technology refresh or an entirely new development effort.) This effect is described in *Software Project Duration and Effort: An Empirical Study* [Barry 2002].

Since the project's customers only have one chance to state their requirements, they are more likely to include every requirement they can think of upfront
[Ching 2004].

Other factors can influence this dynamic. If an acquisition program is expected to be large, even while still in the initial planning phases, it can affect the way that users behave during requirements elicitation. If stakeholders feel that this program is their only shot at change, they'll load the system up with everything they can think of, because there won't be a second chance.

Breaking the Pattern

Once started, the *Longer Begets Bigger* dynamic is as difficult to stop as it would be to stop the inevitable advance of the technological environment that fuels it. If technology obsolescence becomes the issue and the program proceeds using the planned (older) technology, the result will be an immediate technology refresh, or inadequate technology with expensive maintenance.

If the problem is the evolving user needs, the choices are no better. Ignoring those user needs may condemn the system to irrelevance or cancellation because it will not be capable of performing the functions the users need, or of doing them well enough—but choosing to change the system at the users' behest may force the system into another cycle of longer duration and greater investment of effort.

Prevention is the most practical strategy for dealing with the projects—avoiding the dynamic in the first place. Doing so involves several considerations—the anticipated duration of the program, the expected rate of evolution of the needed technologies, and the rate of change of the operational environment. Rapid change calls for smaller, distributed programs rather than large, monolithic systems.

Finally, the identification and implementation of acquisition reforms (e.g., competitive prototyping and improving the corps of acquisition professionals) may ameliorate this dynamic.

5.9 Robbing Peter to Pay Paul

Acquisition programs compete for funding in an environment where any gain achieved in funding for one program often occurs at the expense of another program. Over the long term, this dynamic can significantly unbalance the acquisition process. While initially this is a case of robbing Peter to pay Paul, the “robbery” can produce ripples across a larger set of acquisition programs, perhaps eventually leaving Paul, and others, poorer. While our perspective focuses on the consequences for just one program, clearly the underspend/overspend issue affects the broader acquisition community.

Underspent

This dynamic has its roots in how the government looks at spending. An acquisition program’s rate of spending is monitored almost as closely as the rate of development progress. In fact, during the early stages of a program, spending may be the primary yardstick of success: dollars out the door equals progress. *Underspending*, then, equates with program trouble (whether trouble truly exists or not), and trouble raises the specter of program cancellation, delays, or loss of funding.

Meanwhile, managers of other programs (the Pauls of our allegory) are quite aware of the potential gains they can realize from their colleagues’ underspending. They know just how to reach into Peter’s pockets, and how that can fix their own *overspending* problems.

When a levy comes down, they look across the board and see which programs are not obligating against their goals.

For example, consider what one program leader said: “In FY06 we got our money in March, so there were six months left. But contracts were awarded for 12 months, so [in FY07] we’re still expending FY06 money on those. We’re on track for obligations, but not for expenditures.”

This program, on tap to develop an IT system, was aware of the risks of being underspent. Congress “gave us leeway last year, but this year we’ll have to start doing better,” the

financial manager said. The deputy program manager observed that “If the leadership reviews our expenditures for FY06, we are in danger of losing funding for other task orders.”

Replanning

If a program falls behind on expenditures, it can be targeted as a “bill payer” (the Peter being “robbed” part of our allegory) for another program that’s either short of funding or is considered a priority that deserves additional funding. The bill payer program can lose the underspent portion of its funding. In our example, a team member said his program was eventually designated as a bill payer for \$2 million. The result? At the end of the fiscal year the program’s finance people had to figure out what to do after losing \$2 million. Recalculating the effects of a budget cut can consume weeks—or months—of effort. Worse yet, replanning can happen multiple times, posing a large, ongoing burden to the program.

Performance

The losing program must reset expectations about what it (the bill payer) can deliver, and when. It is not always clear if a funding cut is temporary, if requirements are being removed, or if—in the worst of all possible worlds—the cut is permanent and there is no reduction in system scope.

A common result is that the bill payer program winds up performing poorly compared to its original expectations, while the recipient demonstrates better-than-expected progress. The longer term consequences are predictable. The bill payer doesn't receive its requested funding for the following year, while the beneficiary is fully funded—and may still find a way to use additional unexpected funding reallocations late in the fiscal year.

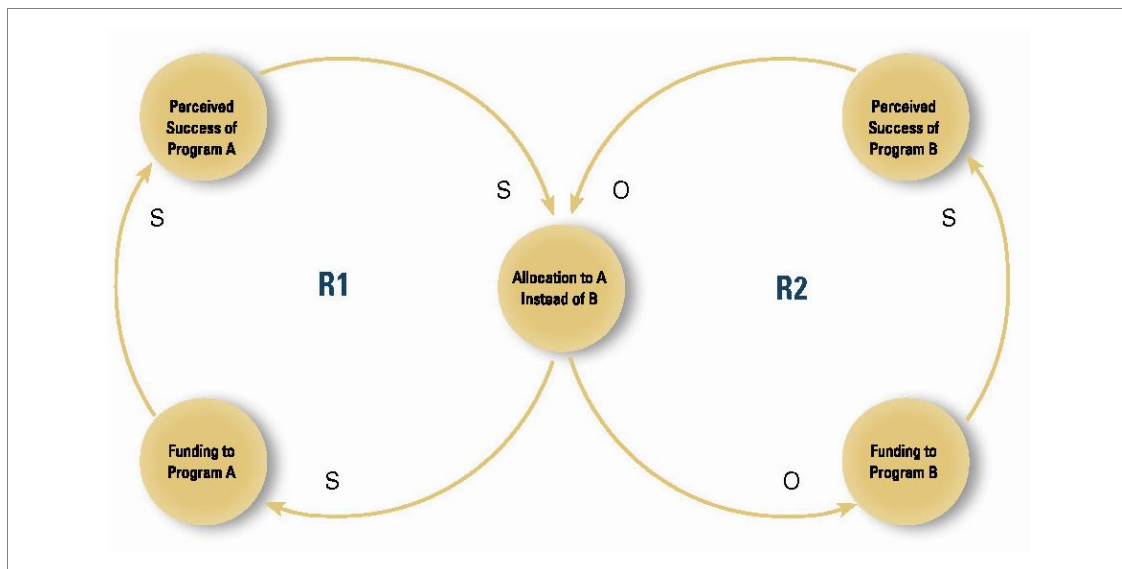


Figure 20: Causal Loop Diagram of "Robbing Peter to Pay Paul"

The Bigger Picture

Acquisition programs compete for funding in what often appears to be a zero-sum game. As Figure 20 indicates, a greater *Allocation to A Instead of B* provides more *Funding to Program A*, which then can exceed progress expectations by spending the additional money, and achieve greater *Perceived Success of Program A*, which then makes it an even *more* likely candidate to receive an additional funding "plus-up" (increment) the next time spending rates are examined.

Robbing Peter to Pay Paul incorporates a bit of self-fulfilling prophecy. The initial suspicion that the more aggressive programs might be better managed, and thus might have a greater likelihood of success, is validated if the program is able to deliver better-than-expected results.

In this situation a program manager with a high-priority program could manipulate this process to his or her advantage by overspending with the expectation that funding will be taken from underspent programs to make up the projected shortfall.

Of course, this is a very dangerous game of "chicken" to play with the sponsor: the program must knowingly overspend without having any guarantee that the acquisition funding process will

come through and deliver the expected additional funds. If it doesn't, the program must either shut down some planned activities or go in search of more funding.

Deliberate overspending occurs as a consequence of how the acquisition system is set up and operates. While unexpected and undesired, overspending can be a by-product of the government's acquisition processes and rules:

1. Money is reallocated mid-year from underspent programs to overspent programs.
2. Program management officers are expected (or given incentives) to act primarily in the best interests of their program, and only secondarily in the best interests of the DoD or the government. The dilemma here is that PMs are simultaneously *expected* to act in the best interests of the PEO, their service, the U.S. armed forces and the U.S. government, but they may only be *given incentives* to act in the interests of their program, which in turn will help to advance their careers.

Breaking the Pattern

Most programs try to deal with this dynamic by playing the game as best they can—trying to keep their spending on plan, and assiduously attempting to avoid the unenviable position of being underspent by whatever means necessary.

To break the dynamic, the primary leverage point is the *Perceived Success of Program A/B*. The program that is designated as the bill payer needs to boost its perceived success, despite having less funding with which to do so, in order to avoid continuing its gradual decline.

Another way for program managers to prevent the *Robbing Peter to Pay Paul* dynamic (aside from keeping spending on plan) is to anticipate the use of the expenditure yardstick to judge program success. The assumption in government and defense acquisition that a program that is spending according to plan is a well-managed program—one that will be successful—is not always valid. Being aware that this assumption is implicit is an important step toward managing its effects and assuring that the organization measures program progress (and potential for success) accurately.

5.10 “Happy Path” Testing

Robustness testing (i.e., “negative” testing) is a standard part of any comprehensive testing approach. It attempts to stress the system by providing “bad” or invalid inputs that the system should either reject, or tolerate gracefully. In this Acquisition Archetype, we look at a project where a development team found itself in crunch mode, and robustness testing took a back seat. Testing instead followed the “happy path”—a tightly scripted process that didn’t duplicate real-world conditions, and only verified that the required functionality was in place and functioning correctly.

Starting Down the Path

The team’s tests of the system did not represent actual operations, because the team did not test real interactions in a realistic environment. Instead, testing followed the “happy path,” verifying that the system came up with the right answers—given the right inputs.

We brought up what happens when everything isn’t right. But the contractor didn’t encourage that kind of testing.

Later, in a review of the project, a government user said that the testing “was all scripted.”

“All of the system test scripts ran fine, but they weren’t real world tests,” the user said. “A couple of us took the privilege of deviating from the scripts, to test [the system] more thoroughly, and see if it would blow up.”

The official test scripts, government users said, always delivered the correct end result. They found that disturbing. “We brought up issues of what happens when everything isn’t right,” said one. “But the contractor didn’t encourage that kind of testing. [The contractor] was adamant that that wasn’t what their testing was for.”

Missing Your Defects and Finding Them Again

None of the initial testing revealed performance as a problem. It became a major issue at the next stage of the program, with a pilot at a single site (the full implementation would eventually encompass many more sites). This real-life test presented a slice of the actual working environment.

“In [initial] testing, they never had problems,” a user said, “and transactions went really fast—they said, ‘Wow!’ it was so fast. When things went live [at the single site], all of the problems started—it was a world of difference in the real performance versus the scripted test performance.”

Another user said the problems they ran into with the single-site pilot occurred because “all of the development work and initial testing was done in the ‘city of Perfect’ ... everything worked perfectly. The problem was with the [real life input] errors and discrepancies, and that caused the problems.”

Robustness testing, and its attempts to break the system by using bad inputs, likely would have revealed the flaws.

Rework...and More Schedule Pressure

The contractor help desk was swamped by the trouble reports that poured in from users in the single site deployment. Developers had to be pulled off new tasks to fix the problems—to do rework.

Putting developers on bug fixing, of course, leads directly to a worsening schedule crunch. All of the rework wasn't planned for, and the program didn't have adequate resources. The team had traveled the Happy Path—but found it was anything but a shortcut. In the end, no one was happy.

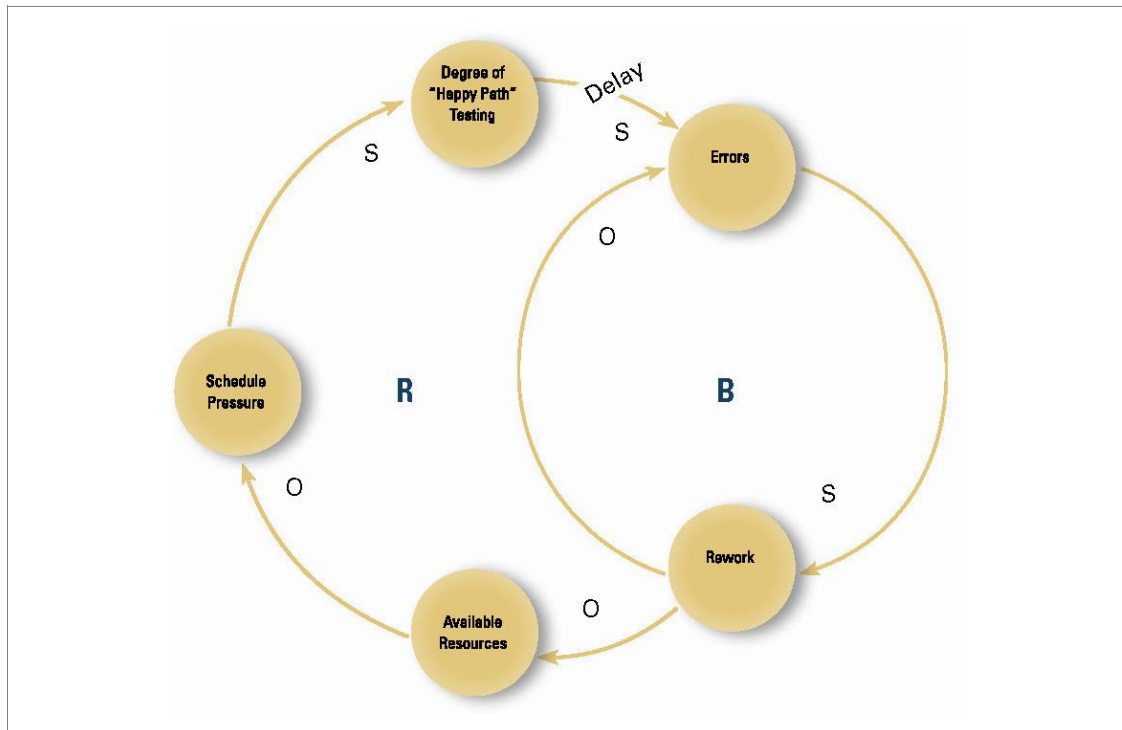


Figure 21: Causal Loop Diagram of “Happy Path” Testing

The Bigger Picture

Testing is a complex activity that, when done properly, employs many different tools and approaches. Testing must be planned and successfully executed on many levels (i.e., unit, integration, and system acceptance) in order to prove that the system is functioning properly before it is deployed. The purpose of scripted testing is to validate that the system is working as expected. Happy Path tests are typically tightly scripted tests of planned system functionality, and are a legitimate strategy for some aspects of testing—but not all. Complete and meaningful testing must also try to emulate the operational environment into which the system will be deployed. Comprehensive testing must attempt to break the system, generating errors in the way that normal users may do when they are using the live system, so that the consequences and probable system behavior can be understood.

All of the system test scripts ran fine, but they weren't real world tests.

Happy Path testing does *not* determine whether the system will behave well in the presence of errors; if used as the primary testing approach, the consequence is that undiscovered problems will still be present in the system. These errors will survive and multiply through successive development phases, and will ultimately be found either very late in development, or by users after deployment—when errors have the greatest impact, and are the most expensive to fix.

Breaking the Pattern

No guide to software testing would advocate Happy Path testing *except* as a single element of a much larger and more comprehensive testing strategy. However, if the program testing budget is inadequate, or the available schedule for testing has been squeezed by prior schedule slips, it may become the only type of testing that can be completed within these constraints.

In trying to break out of the “Happy Path Testing” pattern, the program needs to first acknowledge that the fix they are using—testing only the system functionality that is expected to work—is only mitigating a *symptom* of the actual problem (i.e., abundant system defects).

Next, the program must commit to addressing the *fundamental* problem—finding the defects that will only occur when the system is actually used in the “real world,” or when there are problems in the system’s operating environment.

Several actions can help prevent Happy Path testing:

- Ensure that both resources and schedule are sufficient to provide comprehensive program testing coverage—and that they remain that way throughout the program.
- Require robustness testing that tests system behavior in the presence of input errors, bad data, and problems with the operational environment (network connectivity and similar factors).
- Test entire end-to-end operational scenarios, rather than only specific functions of the system. Problems are more likely to occur when multiple operations are performed in conjunction with one another, rather than in isolation.

5.11 Brooks' Law

Adding manpower to a late software project makes it later.

Brooks' Law is well known in the software engineering community due to the ground-breaking book, "The Mythical Man Month: Essays on Software Engineering" [Brooks 1975].

In this Acquisition Archetype, we look at a program that chose to ignore it, and the consequences of doing so.

Facing an Aggressive Schedule

An information technology claims processing program had fallen behind its cost and schedule goals. A new program manager (PM) was scrambling to meet a strict and fast-approaching deadline imposed by the program's management review board.

Rose-Colored Glasses?

The PM informed the board that the team could not come close to delivering the latest list of requirements for a November release with its current staff of 50.

When board members asked what it would take to meet the deadline, the PM sensed that he had to come up with

a solution on the spot, regardless of how realistic it might be. Stressed and hoping that his program could prove to be the exception to Brooks' Law, the PM proposed having the contractor set up a new, additional development site with 20 to 30 staff.

*We all knew that it
couldn't work.*

It would cost millions of dollars more.

Much later, during an assessment of the program, another manager noted how project stress can force poor decisions. "We bought into the *"mythical man-month,"* the manager said, "even though we all knew it couldn't work."

Belief ... and Doubt

Adding a new site certainly was far from ideal. In addition to the added cost, it introduced increased risk. Some managers later called it "the worst situation we could have"—but they, along with the PM, were committed to the aggressive schedule, and adding the site was the way the PM was allowed to add developers.

Expectations for the new site varied greatly. The PM, who had staked personal reputation on the decision to expand the staff, professed it would speed development, moving the project "50-70 percent ahead." Other team members were less optimistic, believing that in the best case they would be no better off—and might, instead, end up farther behind schedule.

More Work, Not Less

The new site was located in Santa Clara, Calif. It was designed to operate for four months.

Once the program started to add developers and ramp up operations in Santa Clara, however, the effect on the program became apparent. It wasn't good. The re-planning was laborious.

One team lead understood why the expansion had been done and the pressure the PM faced, but observed that "...ramping up impacted the productivity of the original team." This was true despite the technical expertise of the Santa Clara hires.

The added travel and training duties affected the project leads' efficiency. "The leads ... were only able to operate at 50-75 percent of their normal productivity," noted one manager. Along with the ramp-up came "frustration among the team with the long hours," he said.

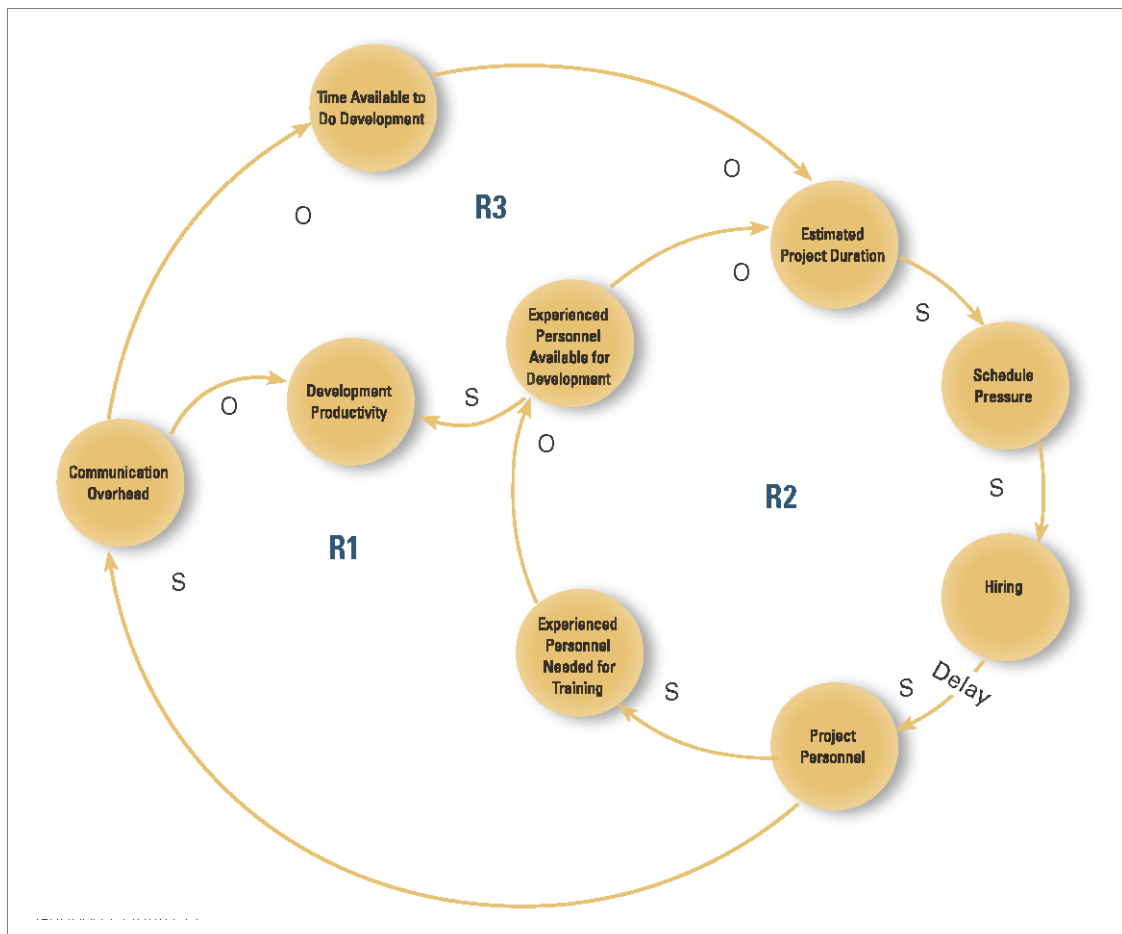


Figure 22: Causal Loop Diagram of "Brooks' Law"

Brooks' Law Wins

Flouting Brooks' Law gained the team nothing except budget overruns.

After opening up the new site and growing the staff by 50 percent, one development manager estimated the team got *half* of the November delivery done.

"If we hadn't brought up the Santa Clara team, we probably would have gotten it done in the same amount of time," he said.

The Bigger Picture

Brooks' Law has been discussed and analyzed extensively in software engineering literature.

The specific behaviors portrayed by the Brooks' Law archetype include the following:

- geometrically increasing communication overhead that
 - reduces development productivity
 - reduces the time available for each individual to do development
- a reduction in experienced personnel available for development (by using them for training of new personnel)

The inner loop of the causal loop diagram (Figure 22) shows the effects of peer training, while the outer loop shows the effects of communication overhead. These new tasks give more work to the already overloaded staff. Assigning these tasks adds coordination and replanning time, and more time is lost to "thrashing" as developers switch between training and development.

If the problem that triggers the *Schedule Pressure* (and seems to require additional manpower) is detected late in the program, an over-reaction is likely. This happens because at this late point comparatively drastic steps must be taken for the intervention to have a chance of working before time runs out. Like other management interventions and improvements, there is a time delay before any benefit will be realized by the program. If the benefit occurs *after* development ends, the program only experiences the negative effect, and the effort is not only in vain, but counterproductive.

The worst-case outcome is that as *Estimated Project Duration* rises even further, there could be *further* increases in *Project Personnel*, requiring another loop through the diagram (Figure 22). However, this spiral happens only if the organization experiencing it is unable to detect the pattern that it is going on (i.e., if they are unable to learn from their experience on the first iteration).

Breaking the Pattern

Adding staff to a late software project is not inherently a bad idea. The circumstances must be right, however. The key is to explicitly recognize and minimize the unintended consequences of adding manpower as shown in Figure 22.

- Adding manpower may be acceptable if there is sufficient schedule in the program to allow it to be done while at the same time meeting the intended system scope. For this reason, it is important to act as early as possible.
- The scale or degree of the added manpower is significant, as a smaller scale increment in staff will help to minimize the explosive increase in communication overhead.
- The experience of the new staff is also critical; domain knowledge and experience with similar systems and the development methodology can significantly reduce the need for training.
- Finally, due to the delay before the new staff will become fully productive, the most financially efficient approach to adding manpower is to amortize the investment by keeping the additional staff on the program after they become fully productive. This is not always possible depending on where the program is in its life cycle, and there may be an explicit acknowledgement that it is more important to try to meet a scheduled delivery date than to be cost-effective.

5.12 Shooting the Messenger

When a program is in trouble, a responsible manager will want to deliver the bad news to upper management. But people are rarely rewarded for this “whistle blowing,” and instead may be ostracized and punished. As is seen in the example below, this has a chilling effect on the other employees and managers, who are then increasingly reluctant to point out any issues, finding it more beneficial to their careers to keep quiet—at least until their tenure with the program has ended.

“Sir, I Have Bad News”

A software development program, underway for several years, repeatedly missed deadlines. So, when the program management declared a firm, “drop dead” delivery date, the team was skeptical. Yet, they kept their reservations to themselves—along with (as the project schedule ticked away) the bad news of continuing setbacks and mounting risks to successful completion of the project.

This reluctance and outright fear to deliver bad news grew out of the team’s experience with what had become an insular, risk-averse organization—an organization where executive management was unwilling to hear from subordinates that their assignments could not be completed, or that a mandated deadline could not be met. Messengers bearing bad news, one team member said, “were shot.”

Executive management was unwilling to hear from subordinates that their assignments could not be completed.

“Voicing problems or raising risks to management made them [the problems] instantly your fault,” he added. “It left you anxious for your position in the organization.”

Fear, Uncertainty, and Doubt

Because the upper management wasn’t willing to hear bad news, the PM became reluctant to identify or escalate risks until they directly affected releases. This meant the risk focus was usually concentrated on dealing with near-crises, contrary to the premise of risk management—proactive identification and mitigation.

That was punishing to the entire staff.

The PM’s view was “very much reactive,” a team member said. The result was that the program could (1) ignore many risks (i.e., “see no evil”) and (2) make decisions that added to the collective risk the program was already facing without having to acknowledge it.

With realistic schedule concerns being ignored, practical alternatives rejected, and having no flexibility to meet the deadline, program management was left with no good options. The PM became resigned to putting on an optimistic face toward the staff, hoping that things would work out by the deadline—despite ample evidence to the contrary.

Of course, in this instance, the drop-dead deadline came and went. High-level meetings were held, more dictates were made—and the program stumbled on.

The Bigger Picture

In many programs facing significant issues, it is not unusual to find that team members defer reporting serious risks or problems to upper management, either out of fear of repercussions or in the hope that local solutions might be found. However, there is nothing more damaging for program management than *not* knowing that there are serious issues lurking under the surface until late in the project schedule.

The PM's view was "very much reactive" ...the program could then ignore many risks that wouldn't have to be escalated to executive management.

If PMO staff members observe that their program is failing in some respect, they generally feel obligated to inform their manager so that the situation can be addressed. However, that is not the case in some organizations—programs where the messenger may not be rewarded for this behavior, but rather punished. Other staff members who observe this result then in turn become increasingly reluctant to point out such issues themselves, finding it more beneficial to their careers to keep their issues to themselves. With these perceived negative consequences, the reluctance to deliver bad news

increases over time. It's also important to note that this pattern doesn't occur on just one side of the program equation—it can happen on the government side as easily as on the contractor side.

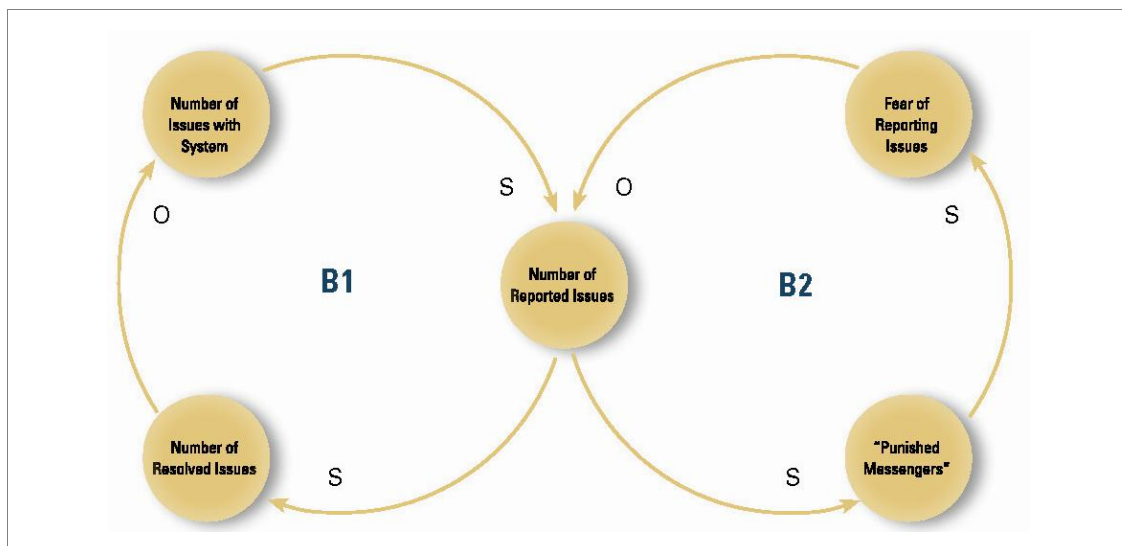


Figure 23: Causal Loop Diagram of "Shooting the Messenger"

This archetype shows how the behavior of punishing the messenger postpones acknowledgement of more risks into the future, which has direct implications for risk management. Although it is more effective to mitigate risks and prevent them from becoming problems, programs are more likely to work to resolve actual *problems* (perhaps because they can no longer be avoided). Risks, on the other hand, may be ignored until they *become* problems—but to the detriment of the program.

An important related aspect is that managers may find it more beneficial to keep a program alive until their tenure with it has ended. The comparatively short-term assignments given to program managers in the Department of Defense make simply leaving the program at the end of a tour an acceptable way to “solve” the problems: they become someone *else’s* problems.

Breaking the Pattern

“Shooting the Messenger” could be attributed to a lack of integrity—or it could be viewed as an indirect consequence of the larger system environment. The archetype diagram (Figure 23) shows ways of breaking the overall reinforcing loop at the points where the links can be controlled.

For example, it would be feasible to break the link between *Punished Messengers* and *Fear of Reporting Issues* only if the punishment was insignificant. The only practical point to break out of this dynamic is between *Number of Reported Issues* and *Punished Messengers*—in other words, in the actions of upper management. If this dynamic is entrenched behavior in an organization, it is very difficult to change, because reporting bad news requires trust, and trust is in short supply in such organizations.

A credible signal must be sent to the staff that things are different now, such as the institution of an established risk management process. Once a risk or issue is reported without adverse consequences, trust will begin to build again and the flow of honest information will eventually resume.

To prevent this dynamic, some advocate the independence of status reporting from decision making, “because that function is unlikely to provide accurate status if the current status happens to be unfavorable/unflattering to the PMO” [Flowers 1996]. Program managers aren’t put in the position of feeling they must suppress bad news about the program to protect their careers.

6 Challenges, Implications, and Future Directions

There is no doubt that the acquisition of software-intensive systems is a difficult and complicated undertaking. Acquisition organizations are examples of dynamic systems, where the interactions between the PMO, the contractor (prime), subcontractors, sponsors, and users are complex and nonlinear—producing behavior that appears unpredictable and unmanageable. To summarize, software intensive acquisition is challenging for the following reasons.

There can be complex interactions between the PMO, contractors, sponsors, and users.

- The full chain of actions and their longer term consequences are not clear.
- It is hard to apply corrective actions when the situation's status is uncertain.

Significant delays exist between applying changes and seeing results.

- It is inherently difficult to control systems with long delays between cause and effect; for example, steering an aircraft carrier or reorganizing a large company or department.

Progress and results are often unpredictable and unmanageable.

- There is limited visibility into real progress and status.
- The complexity of the interdependencies has unintended consequences.

There can be uncontrolled escalation of situations despite best management efforts.

- Misaligned goals can drive potentially conflicting behaviors.

Linear partitioning is the standard approach to address large systems.

- When systems have feedback between components that are partitioned, it makes it difficult to see and address these interactions.

There is exponential growth of interactions as size grows linearly.

We have observed that in traditional approaches to analyzing and resolving problems, problems are viewed linearly—occurring in a sequence—and they are decomposed into smaller pieces in a “divide and conquer” approach. However, such approaches fail if the system is nonlinear, with “downstream” aspects “feeding back” and affecting “upstream” components, such as is the case with acquisition. Similarly, a “divide and conquer” approach is flawed when there are significant and complex interactions between the components that are being partitioned; the partitioning *itself* makes it impossible to see and address these interactions.

Stories of acquisition failures and software development failures abound; they are not new, nor are they in short supply. Analysis of these failures has been an ongoing activity for many years, albeit with varying degrees of comprehensiveness and success. Thus, many of the acquisition concepts presented here are neither revolutionary nor novel. The question then arises as to what value this work can bring to the software acquisition and development community. We have tried

to demonstrate how systems thinking can help to identify dysfunctional (counterproductive or self-destructive) behaviors and offer insight into interventions to manage, stop, and prevent such behaviors.

The purpose of this effort was to achieve three objectives:

1. Provide the software acquisition community with a set of quintessential archetypes (based on real-world experience) that reflect some of the typical dilemmas, tradeoffs, and interdependencies evident in everyday acquisition practice.
2. Present this as learning-in-action through clear and understandable stories, along with compelling analysis.
3. Provide practical guidance on how to identify, break out of, and prevent these counterproductive behaviors.

To the extent that this report has met these objectives, it can be a useful aid to software acquisition practitioners who must confront these behaviors and patterns in their work. While the set of software acquisition and development archetypes presented here is by no means complete or definitive, these archetypes are representative of the types of problems seen in software acquisition. We hope that these archetypes will resonate with software acquisition practitioners as being (unpleasantly) familiar. Once identified, these archetypes should become substantially easier to recognize and more tractable to manage. It is the authors' hope that acquisition practitioners will actively look for signs of these archetypes in their activities and workplaces and apply some of the suggested techniques when addressing them.

Finally, because this report presents an initial set of software acquisition and development archetypes, it should serve as a starting point for additional research. There are many more archetypes that can be identified, described, and collected. The authors hope to explore this area further, to define additional archetypes and solution elements that might be brought to bear. This report has presented the results of research into the use of systems thinking to understand the behavior of software acquisition programs. There are some significant challenges and implications that can be drawn from the software acquisition and development archetypes developed for this work. These are discussed in the following sections.

6.1 Short-Term Thinking

Program management on either the government or the contractor side is at the heart of many of the acquisition and development archetypes presented here [GAO 2004]. While these 12 archetypes do not constitute a representative sample, virtually all of them require the active involvement of program management to be realized. There is a recurring theme among many of these archetypes where the tactical short-term fix (or view) is chosen over the strategic long-term view. In improvement planning, this is often justified and applauded when we pursue the “low-hanging fruit.” However, “low-hanging fruit” and “Fixes That Fail” are often synonymous, except that “low-hanging fruit” sounds positive and efficient, whereas “Fixes That Fail” sounds negative and defeatist. It is difficult for employees to take the long-term view and work the fundamentals when the organizational culture rewards actions that appear to produce quick, positive results, even if they turn out to be “Fixes That Fail” in the longer term. There may also be pressure to avoid looking for unintended consequences when short-term solutions are being promoted.

Finally, in the best of circumstances, we must acknowledge that patterns and structural properties are hard to perceive and discern. Our organizations are full of situational flux, and often there are no incentives to look at events and activities, either closely, broadly, or over time.

In game theory there is a famous paradox that is referred to as “The Prisoner’s Dilemma.” Originally posed by Merrill Flood and Melvin Dresher of RAND, The Prisoner’s Dilemma was later formalized by Princeton mathematician Albert Tucker [Prisoner’s Dilemma 2009]. The Prisoner’s Dilemma closely resembles the “Tragedy of the Commons” in that in both cases people act in their own self interest at the expense of others, and even at the expense of their *own* long-term personal interests. This manifests itself as short-term thinking because, other than financial planning for their children or their own retirements, people rarely have a long-term time horizon in mind with respect to their own careers, much less a 10-20 year acquisition program. People are motivated to do what’s best for them personally, and their decisions reflect that self-interest in the short term.

This kind of thinking appears with alarming regularity and is seen in such systems archetypes as “Fixes That Fail,” “Shifting the Burden,” “Drifting Goals,” and the “Tragedy of the Commons.” In the context of software acquisition and development, we’ve discussed several archetypes that illustrate this bias, and there are many other instances, including the following.

- “Bow Wave Effect,” or repeatedly postponing the most difficult development work

- “Happy Path” testing, one instance of a larger pattern of cutting corners on quality processes, such as excessively shortening the design phase, eliminating code reviews, etc.

- “Staff Burnout and Turnover,” or relying on increased pressure (and individual heroics) instead of good processes

- Inappropriate incentives that reward highly visible attributes such as on-time delivery, while subtly sacrificing quality to achieve it

- Short rotations of program managers through acquisition programs, removing hard-earned experience as soon as it has been gained

- Neglecting preparation for long-term software sustainment issues (e.g., gaining the rights to, and ensuring compatibility with, the contractor’s software development environment)

To address this situation the government acquisition community needs to take steps to better align the personal goals of PMO staff with the long-term objectives of government acquisition. Without this alignment the government acquisition system will continue to rely largely on the integrity of the participants while at the same time undermining that very integrity. This is discussed in additional detail in the next section.

6.2 Misaligned Goals

As was discussed in the “PMO Versus Contractor Hostility” archetype, one reason behind short-term thinking is misaligned goals. This can occur at a variety of levels in software acquisition—between the program manager (as an individual) and the program, between the PMO and the contractor, between the PMO and the program executive office (PEO), between the PEO and the service, and so on. In the case of the government PMO and the contractor, the PMO generally prefers lower costs (*do more with less money*) and shorter schedule (*field the system sooner*), but the contractor prefers higher cost (*more profit*) and a longer schedule (*more stable work force*). Another example would be the misalignment of goals between the user (or user representative) and the program, in which the user wants all of the capability possible from the system because, unlike the buyer of a house, the user isn’t paying for the system, so price isn’t a significant consideration. This leads to longer schedules and cost overruns, but these concerns may not be critical to the user representative and so may be an insignificant disincentive to rein in user demands. Requirements scope creep would be less of an issue in acquisition if all the stakeholders had the same level of interest in reducing cost that the sponsor does.

Misaligned goals commonly occur in the absence of adequate governance, where governance is the set of rules that control an activity and the rewards or punishments for the participants. The underlying principle is that unless laws, policies, or other regulations encourage them to do otherwise, people tend to behave in their own best interests (i.e., rational self-interest, the basic assumption behind much of modern economics theory). Misaligned goals are connected to short-term thinking when (for example) contractor goals conflict with program goals, and thus contractor self-interest drives decision making that may not be in the best interest of the program. Misaligned goals play a part in each one of the examples discussed above that relate to short-term thinking:

In “The Bow Wave Effect” postponing the most difficult development work can be the result of the contractor wanting to create a better near-term perception of progress in the eyes of the PMO, or of the PMO trying to create a good impression for those attending key program reviews—achieving progress goals (and rewards of additional funding) at the expense of the longer term goal of program success.

In “‘Happy Path’ Testing” the contractor’s goals of passing system testing may be met, but at the expense of the goal of successful system deployment.

In “Staff Burnout and Turnover,” the goals of meeting near-term deliverables may be met through sustained schedule pressure, but at the cost of losing a significant portion of the development team, thus substantially jeopardizing any further progress.

Incentives that reward only on-time delivery are rewards for goals that may not be properly aligned with the overarching goal of delivering the highest quality system if quality is not explicitly rewarded as well—and thus is quietly sacrificed in order to achieve the reward for on-time delivery.

Short rotations of program managers through acquisition programs mean that the program manager has little incentive to fund and support significant activities that will not bear fruit during his or her tenure on the program, and thus will have little effect on their future career or personal goals.

Neglecting preparation for software sustainment can occur when no incentives are provided to perform work that will 1) go largely unnoticed for many years (and therefore not help the personal or career goals of PMO staff who are responsible for them) and 2) only be of value long after the individual has left the program.

A careful review of the alignment of goals across the different levels of the government acquisition system would be helpful. It would expose opportunities for the improvement of governance in the form of providing rewards and incentives that could bring the goals of the different parties into better alignment and reduce the degree of dissonance that currently exists among the stakeholder groups.

6.3 Future Directions

During the course of this research it became apparent that three disciplines were related to the application of systems thinking and use of the systems archetypes: system dynamics, game theory, and chaos theory. All three have been applied to greater or lesser extent to the analysis of management and organizations, and research continues to be active in each area. These topics are presented here as potentially fruitful areas for future research in applying systems thinking to software development and acquisition organizations.

6.3.1 System Dynamics

Both system dynamics and systems thinking require a clear understanding of the model(s) behind the system, where systems thinking requires a higher level qualitative understanding and system dynamics requires a more detailed quantitative understanding. Because the systems of interest are complex and nonlinear, their study defies traditional mathematical analysis. For example, the effects of gravity are often viewed as being linear, but really the interactions of multiple moving bodies with intersecting gravitational fields must be analyzed. While the moon orbits the Earth, the Earth is simultaneously orbiting the moon as well. This is a problem that is now *nonlinear* and very complex. Computer simulation is often used as the only quantitative method for analyzing such problems (because of the complexity of nonlinear feedback), which is why it is an essential component of system dynamics work.

One of the primary differences between system dynamics and systems thinking is in the visibility of the model. In system dynamics, the *behavior* of the model can be made as clear in terms of the outcomes (through simulation) as it would be if observed in real life—the actual underlying *structure* of the model is not easily visible. When it is made visible, it is presented in stock-flow diagrams with hidden mathematical relationships that are often too complex for anyone other than the original modelers to comprehend. Systems thinking, however, is about explaining and understanding the interrelationships that the model is based upon for the benefit of those who are part of, or must interact with, the system. Despite the issues with attempting to accurately

characterize all of the interrelationships within a social system such as an acquisition organization, the quantitative approach of system dynamics, along with the computer's ability to manage the complexity of simulating large system models, makes system dynamics an attractive approach for helping to derive high-level qualitative conclusions about the behavior of complex organizational systems.

6.3.2 Game Theory

There are many parallels and overlaps between game theory and systems thinking, as well as system dynamics. One example is the “Tragedy of the Commons” structure, which is described in literature as both a classic game theory game and a systems archetype [Hardin 1968]. The “Tragedy of the Commons” is essentially a multiple player version of the classic Prisoner’s Dilemma game because while the popular (i.e., “dominant”) strategy is to *defect*, or *not* to do your fair share, this is ultimately a losing strategy for the larger community. The intent of any larger organization is to avoid outcomes in which people optimize their decisions and their behavior solely for their own personal outcomes, rather than for the broader outcomes of the organization. The only effective solution to either the “Tragedy of the Commons” or the Prisoner’s Dilemma is to provide governance in the form of rewards and incentives that will avoid the underlying trap in which individually optimal decisions lead to collectively inferior solutions.

Another example of the overlap between the two disciplines is the game theory game referred to as mutually assured destruction, which describes a nuclear deterrence strategy and illustrates the “Escalation” dynamic both in the response of the “players” in the game and in the arms race required by the players to maintain parity in nuclear capability in order to keep the strategy in place.

There have been formal calls to increase the amount of research being done on the connections between system dynamics and game theory. In March 2007 Qifan Wang, the president of the System Dynamics Society, stated that “integrating system dynamics with game theory will allow us to better explore many major social, political, economic, ecological and other problems our world faces” [Wang 2007]. This direction may prove to be an important addition to the set of analytical tools available for studying complex systems.

6.3.3 Chaos Theory

Chaos theory is defined as the study of a wide range of complex nonlinear dynamic systems [Gleick 1987]. These systems are not truly random, but they do have deterministic rules and include interactive and nonlinear feedback relationships between the variables in the system. Increasingly, organizations are seen by researchers as being nonlinear dynamic systems. As chaos theory is applied to the study of organizations it is believed that the interactions among the comparatively simple behaviors of individuals produce the complex overall behaviors of organizations. It is the nonlinear aspect of the resultant system that gives the appearance of chaos. This means that the long-term results (and side-effects) of the actions of an organization cannot be predicted with any more certainty than Lorenz was able to predict long-term weather

patterns. This exemplifies the key characteristic of chaotic systems, namely sensitivity to initial conditions in which the outcome may vary greatly as a result of seemingly insignificant differences in the starting point(s). It is an ongoing research effort to determine whether the behavior of organizations is truly an instance of chaos theory or is illustrating another, even more subtle pattern. However, the analogy has proved useful thus far in conceptualizing management theory, and the continued pursuit appears to be promising.

References

URLs are valid as of the publication date of this document.

[Abdel-Hamid 1991]

Abdel-Hamid, T. K. & Madnick, S. E. *Software Project Dynamics: An Integrated Approach*. Prentice Hall, 1991 (ISBN: 0138220409). www.pearsonhighered.com/bookseller/product/Software-Project-Dynamics-An-Integrated-Approach/9780138220402.page

[Barry 2002]

Barry, Evelyn J.; Mukhopadhyay, Tridas; & Slaughter, Sandra A. "Software Project Duration and Effort: An Empirical Study." *Information Technology and Management* 3 (2002): 113-136. www.springerlink.com/content/pg8650012871658p/

[Brooks 1975]

Brooks, Frederick. *The Mythical Man Month*. Addison-Wesley, 1975 (ISBN: 0201835959).

[Ching 2004]

Ching, Clarke. "The Software Project Manager's Conflict—to allow, or not to allow, change." MBA diss., Open University, April 2004.

[Dews 1979]

Dews, Edmund; Smith, Giles K.; Barbour, Allen; Harris, Elwyn; & Hesse, Michael. *Acquisition Policy Effectiveness: Department of Defense Experience in the 1970s* (R-2516-DR&E). A report prepared for the Undersecretary of Defense for Research and Engineering. The RAND Corporation, October 1979.

[Drummond 1996]

Drummond, Helga. "The politics of risk: trials and tribulations of the Taurus project." *Journal of Information Technology II* (1996): 347-357.

[Evans 2000]

Evans, Phillip & Wurser, Thomas S. *Blown to Bits: How the New Economics of Information Transforms Strategy*. Harvard Business School Press, 2000.

[Flowers 1996]

Flowers, Stephen. *Software Failure: Management Failure*. John Wiley & Sons, 1996 (ISBN: 0471951137).

[Ford 2005]

Ford, David N. & Taylor, Tim. "Why Good Projects Go Bad: Managing Development Projects Near Tipping Points." *Proceedings of the 23rd International Conference of the System Dynamics Society*, July 2005.

[Forrester 1971]

Forrester, Jay Wright. *Principles of Systems*. Pegasus Communications, 1971 (ISBN: 1883823412). www.pegasuscom.com

[GAO 2004]

Government Accountability Office. *Defense Acquisitions: Stronger Management Practices Are Needed to Improve DoD's Software-Intensive Weapon Acquisitions* (GAO-04-393), March 2004.

[Gleick 1987]

Gleick, James. *Chaos: Making a New Science*. Penguin Books, 1987 (ISBN: 0140092501).

[Haraldsson 2005]

Haraldsson, Hördur; Sverdrup, Harald; & Belyazid, Salim. "The Tyranny of Small Steps I: Discovery of an Archetypical Behaviour." *Proceedings of the 23rd International Conference of the System Dynamics Society*, 2005.

www.systemdynamics.org/conferences/2005/proceed/proceed.pdf

[Hardin 1968]

Hardin, Garrett. "The Tragedy of the Commons." *Science* 162, 3859 (December 1968): 1243-1248. www.sciencemag.org/cgi/content/full/162/3859/1243

[Kim 1993]

Kim, Daniel H. *System Archetypes: Diagnosing Systemic Issues and Designing High-Leverage Interventions*. Pegasus Communications, 1993.

[Kim 1998]

Kim, Daniel H. & Anderson, Virginia. *Systems Archetype Basics: From Story to Structure*. Pegasus Communications, 1998 (ISBN: 1883823188). www.pegasuscom.com

[Lorenz 1963]

Lorenz, Edward N. "Deterministic Nonperiodic Flow." *Journal of the Atmospheric Sciences* 20, 2 (March 1963): 130-141. http://eapsweb.mit.edu/research/Lorenz/Deterministic_63.pdf

[Merle 2005]

Merle, Renae. "New Pentagon Panel Reviews Acquisition." *The Washington Post*, July 18, 2005. www.washingtonpost.com/wp-dyn/content/article/2005/07/17/AR2005071700717.html

[Myers 2002]

Myers, Margaret. "Power to the Edge: Transformation of the Global Information Grid." Edited from a presentation given by Margaret Myers, Principal Director, Deputy Chief Information Officer, Department of Defense, 2002.

[Prisoner's Dilemma 2009]

Stanford Encyclopedia of Philosophy. "Prisoner's Dilemma," 2009. <http://plato.stanford.edu/entries/prisoner-dilemma>

[Rahn 2005]

Rahn, Joel, R. “Fear and Greed: A Political Archetype.” *Proceedings of the 23rd International Conference of the System Dynamics Society*. Boston, MA, 2005.

[Repenning 2001]

Repenning, Nelson P.; Goncalves, Paulo; & Black, Laura J. “Past the Tipping Point: The Persistence of Firefighting in Product Development.” *California Management Review*, July 1, 2001.

[Richardson 1986]

Richardson, George P. “Problems With Causal Loop Diagrams.” *System Dynamics Review* 2, 2 (1986).

[Senge 1991]

Senge, Peter. *The Fifth Discipline: The Art and Practice of the Learning Organization*. Doubleday, 1991 (ISBN: 0385260954).
www.randomhouse.com/catalog/display.pperl?isbn=9780385517256

[Senge 1994]

Senge, Peter. *The Fifth Discipline Fieldbook: Strategies and Tools for Building a Learning Organization*. Doubleday, 1994 (ISBN: 9780385472562).
www.randomhouse.com/catalog/display.pperl?isbn=9780385472562

[Wang 2007]

Wang, Qifan. “From the President.” *System Dynamics Newsletter* 20, 1 (March 2007).
www.systemdynamics.org/newsletters/2007/07-03newsltr.htm

[Wolstenholme 2003]

Wolstenholme, Eric. “Towards the definition and use of a core set of archetypal structures in system dynamics.” *System Dynamics Review* 19, 1 (2003): 7-26.

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE September 2010		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Success in Acquisition: Using Archetypes to Beat the Odds			5. FUNDING NUMBERS FA8721-05-C-0003	
6. AUTHOR(S) William E. Novak & Linda Levine				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2010-TR-016	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116			10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-2010-016	
11. SUPPLEMENTARY NOTES				
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			12B DISTRIBUTION CODE	
13. ABSTRACT (MAXIMUM 200 WORDS) This project on patterns of failure is based on experiences with actual programs and employs concepts from systems thinking to analyze dynamics that have been observed in software development and acquisition practice. The software acquisition and development archetypes, based in part on the general systems archetypes, have been created as part of an ongoing effort to characterize and help manage patterns of counterproductive behavior in software development and acquisition. This report introduces key concepts in systems thinking and the general systems archetypes, and then applies these concepts to the software-reliant acquisition domain. Twelve selected software acquisition and development archetypes are each described and illustrated by a real-life scenario, and guidance is provided on both recovering from and preventing these dynamics. Finally, the authors consider implications of the work and future directions for research.				
14. SUBJECT TERMS acquiring software, acquisition, acquisition archetypes, acquisition patterns of failure, archetype, patterns of failure, software acquisition, systems thinking			15. NUMBER OF PAGES 94	
16. PRICE CODE				
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

